

由浅入深PostgreSQL

[奥]汉斯·尤尔根·舍尔希 ◎著
彭煜玮 ◎译

从菜鸟到PostgreSQL数据库高手实用宝典

- 高级数据库设计理念
- 检索与查询优化
- 事件驱动
- 并发事务
- 表分区
- SQL和服务器调优
- 服务器维护与监测
- 复制与恢复
- 高可用性
- 常见与不常见的故障排除
- 高级管理任务
- 从MySQL和Oracle迁移到PostgreSQL

清华大学出版社

由浅入深 PostgreSQL

[奥] 汉斯·尤尔根·舍尔希　著
彭煜玮　译

清华大学出版社
北　京

内 容 简 介

本书从一位资深 PostgreSQL 专家在多年咨询、技术支持工作中的切身体会出发，深入介绍了开源数据库管理系统 PostgreSQL 9.6 版本中的主要特性，其内容涵盖了作为一个 PostgreSQL 数据库从业人员经常会接触到的主题：事务和锁定、索引的使用、高级 SQL 处理、日志文件和统计信息、查询优化、存储过程、安全性、备份与恢复、复制、各类扩展、故障排查、系统迁移。作者通过亲身经历和直观的例子，详细介绍了 PostgreSQL 主要特性的工作原理、常用配置以及常见的误区，是一本实用性很强的 PostgreSQL 进阶指南，能帮助有一定 PostgreSQL 知识的读者深入了解 PostgreSQL 中更多更全面的高级特性。

本书适合数据库管理人员和开发人员了解和学习 PostgreSQL。通过阅读本书，读者可以对 PostgreSQL 有一个全面透彻的了解。

北京市版权局著作权合同登记号 图字：01-2017-6581

图书在版编目（CIP）数据

由浅入深 PostgreSQL/（奥）汉斯・尤尔根・舍尔希著，彭煜玮译. —北京：清华大学出版社，2018
书名原文：Mastering PostgreSQL
ISBN 978-7-302-51288-2

Ⅰ.①由… Ⅱ.①汉… ②彭… Ⅲ. ①关系数据库系统 Ⅳ.①TP311.138

中国版本图书馆 CIP 数据核字（2018）第 220273 号

责任编辑：贾小红
封面设计：刘　超
版式设计：魏　远
责任校对：马子杰
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：北京鑫海金澳胶印有限公司
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张：**22　　**字　　数：**451 千字
版　　次：2018 年 10 月第 1 版　　**印　　次：**2018 年 10 月第 1 次印刷
定　　价：98.00 元

产品编号：077167-01

关 于 作 者

Hans-Ju□rgen Schönig 拥有 18 年的 PostgreSQL 经验，是一家名为 Cybertec Schönig & Schönig GmbH（www.postgresql-support.de）的 PostgreSQL 咨询和支持公司的 CEO。该公司已经成功地为全球数不尽的客户提供了服务。

在 2000 年创建 Cybertec Schönig & Schönig GmbH 之前，他是一家专注于奥地利劳动市场的私营调查公司的数据库开发人员，当时他的主要工作是数据挖掘和预测模型。他已经写了好几本有关 PostgreSQL 的书。

关于审稿人

Shaun Thomas 从 2000 年年末开始从事 PostgreSQL 的工作。从 2011 年开始，他已经成为 PostgresOpen 大会的常客，并且在大会上多次发表了关于如何处理极限吞吐、高可用性、监控、架构和自动化方面的演讲。他贡献了一些 PostgreSQL 扩展以及一种管理大规模数据库集群的工具。有时，他也在本地大学里担任客座讲师。他的目标是帮助社区把 PostgreSQL 打造成一种更大、更好的数据库，让每个人都乐在其中。

www.PacktPub.com

您购买的书籍的支持文件和下载请访问 www.PacktPub.com。

您是否了解 Packt 为每一本出版的书籍都提供了电子书版本（有 PDF 和 ePub 文件可用）？您可以在 www.PacktPub.com 上升级为电子书版本，并且作为一位实体书客户，您可以在电子书备份上享受到折扣。详情请通过 service@packtpub.com 与我们联系。

在 www.PacktPub.com 上，您还可以阅读一些免费技术文章，请注册一系列免费通信，这样可以接收 Packt 实体书和电子书的专享折扣和特供。



https://www.packtpub.com/mapt。

用 Mapt 获得最受欢迎的软件技巧。Mapt 让您能获得对所有 Packt 书籍和视频课程的完全访问，还有业内领先的工具帮助您规划个人发展并促进您的事业。

为什么要订阅?

- 完整访问 Packt 出版的每一本书籍。
- 复制粘贴、打印内容以及对内容加书签。
- 通过 Web 浏览器即时访问。

客 户 反 馈

感谢购买这本由 Packt 出版的书。在 Packt，质量是我们编辑处理的中心任务。为了帮助我们提高，请在本书的 Amazon 页面 https://www.amazon.com/dp/1783555351 上留下您中肯的评价。

如果您愿意加入我们的正式评阅团队，您可以发邮件到 customerreviews@packtpub.com。我们将向正式评阅人回馈免费的电子书和视频来奖励他们有价值的反馈。请毫无保留地帮助我们提高我们的产品！

译　者　序

PostgreSQL 是一种历史悠久的开源对象关系型数据库管理系统，它不仅支持关系型数据库的各种功能，而且还具备类、继承等对象数据库的特征。这个起源于加州大学伯克利分校（UCB）的数据库研究计划，目前已经衍生成一项国际开发项目，并且拥有越来越广泛的用户群以及越来越多的应用案例。

本书根据一位资深 PostgreSQL 专家多年的从业经验，深入介绍了开源数据库管理系统 PostgreSQL 的主要特性。其内容涵盖了数据库用户日常工作中经常会接触到的话题，包括并发控制、索引、高级 SQL 处理、日志和统计、查询优化、存储过程、安全性、高可用等。对于日益增长的 PostgreSQL 用户群体来说，这是一部难得的好书。

在本书的翻译过程中，除译者之外，彭钰杰、段芳、段亮圩、兰海、韩珂、郇凌翔、吴瑕、李宇珺、何子龙、段辉银、王淞、赖思超、龙德秀、罗倩雯、徐明民、王敏、岳名亮、汪佳、彭文汉、熊春娥等人也参与了本书的翻译工作，在此一并表示感谢。

限于译者的水平，译文中难免有错误和不妥之处，恳请广大读者批评指正。

彭煜玮

2017 年 12 月 17 日于珞珈山

推 荐 序 一

从互联网、移动互联网到互联网+，再到 IoT、万物互联、人工智能，只用数十年计算机就从科技行业渗透到了几乎所有的传统行业：线上线下打通，甚至将来会完全融合。由于企业数据爆炸性增长，数据库在整个变革中承担了非常重要的角色，掀起了数据驱动的狂潮。

PostgreSQL 数据库凭借丰富的功能、强大的扩展接口能力、稳定的数据库内核正风靡全球，在互联网、金融、物联网、传统企业等行业占据越来越重要的地位。全球数据库权威评测网站 DB-Ranking 的评测结果来看，PostgreSQL 最近几年在所有数据库产品中发展是最为迅猛的，同时新出的 PG 相关书籍已达数百本之多。

本书作者是具有 18 年 PG（PostgreSQL）数据库经验的国外大牛，本书的内容非常丰满。（1）原理讲解深入浅出，将官方手册中较难理解的内容简单明了地展示给读者，适合 DBA、架构师阅读。（2）在 SQL 部分给读者讲解了 SQL 高级用法、存储过程的开发，掌握这些用法可以在实际工作中大大降低开发工作量，轻松实现较高难度的业务需求，建议开发者、数据分析师阅读。（3）讲解了数据库安全、备份恢复、排错、优化，适合 DBA 阅读。（4）PostgreSQL 扩展接口丰富，本书在数据库扩展包部分给读者展示了常用的扩展包，以及扩展包的开发方法，适合开发者、架构师阅读。（5）本书还包含了迁移部分，帮助读者了解如何从异构数据库（如 Oracle、SQL Server 等）、同构数据库迁移到 PostgreSQL。

译者彭煜玮先生，武汉大学计算机学院副教授，是我多年的良师益友，同时也是 PostgreSQL 中国社区的核心成员。他培养输出的数据库人才不尽其数，他们分布在阿里巴巴、华为、腾讯等企业，推动了中国数据库行业的发展。

期待新书早日面世，加快 PostgreSQL 在国内开发人员、DBA、数据分析师、架构师等从业人群中的普及，将其应用到更多的企业，实现技术为业务服务的转化。

周正中
阿里云高级技术专家
2018 年 9 月

推荐序二

数据库技术出现于 20 世纪 60 年代中后期，堪称电子计算机领域最古老的技术之一。自诞生至今，数据库几乎是每一个“严肃”应用不可或缺的一部分。在这几十年的演进过程中，数据库领域发生过很多“有趣”的技术对抗，包括网状模型与关系模型之争、关系数据库与 XML 数据库之争、NoSQL 与 SQL 之争等。直到现在，数据库技术还是最前沿的计算机技术之一，无论工业界还是学术界，数据库相关的研究和开发都非常活跃。近十年，伴随着大数据处理需求的发展，数据库领域出现多种探索和变迁，某些趋势逐渐明朗起来。趋势之一是 SQL 作为大数据处理的“万向头”被广泛接受，这可以从很多 NoSQL 开始支持 SQL 得到印证；另外一个趋势是向事务型、分析型混合处理（HTAP）发展。

PostgreSQL 是最成熟最先进的开源 SQL 数据库之一，其 OLTP 处理能力强大。近几年在 OLAP 方面也有长足发展，包括多核优化、并行处理、分区表等特性。PostgreSQL 扩展能力强大，通过扩展可以处理各种各样的数据，譬如流式数据、时间序列数据、地理信息数据等。还有很多项目以 PostgreSQL 为基础构建分布式数据库。总而言之，PostgreSQL 功能强大、社区活跃、发展前景广阔，是一款很值得投资学习的数据库。

本书全面介绍了 PostgreSQL 数据库日常进阶管理的各个方面，提供了大量的实例以方便读者实践。此外作者还阐述了很多实战中总结的心得和技巧。译者具有多年的 PostgreSQL 教学、实操和内核开发经验，对原书理解透彻，语言流畅准确。感谢作者和译者为 PostgreSQL 社区贡献这样一本出色的书籍。

“至哉天下乐，终日在几案”，祝大家阅读愉快。

姚延栋

Pivotal 中国 Greenplum 研发总监

2018 年 9 月

前　　言

PostgreSQL 是一种开源数据库管理工具，它可以被用于处理大型数据集（大数据）并且可以被用作一种 JSON 文档数据库。它也在软件和 Web 领域中有很多应用。本书将让读者能够构建更好的 PostgreSQL 应用并且更有效地管理数据库。

本书所涵盖的内容

第 1 章 PostgreSQL 概述，使读者从总体上了解 PostgreSQL 及其特性。读者将学到 PostgreSQL 中可用的新事物和新功能。

第 2 章 理解事务和锁定，将涵盖任意数据库系统最重要的方面之一。没有事务的存在，数据库通常无法正确地工作。因此理解事务和锁定对于性能以及专业工作来说都是很关键的。

第 3 章 使用索引，涵盖了读者需要了解的与索引有关的方方面面。索引是性能的关键因素，因此它是获得良好用户体验和高吞吐量的重要基石。索引的所有重要方面都会在本章中被涵盖。

第 4 章 处理高级 SQL，将介绍现代 SQL 的一些最重要的概念。读者将学到窗口函数以及其他重要的更现代化的 SQL。

第 5 章 日志文件和系统统计信息，将引导读者通过更多管理任务，例如日志文件管理和监控。读者将学到如何观察其服务器并且从 PostgreSQL 中提取运行时信息。

第 6 章 优化查询获得良好性能，将告诉读者所有有关良好 PostgreSQL 性能的内容。本章将涵盖 SQL 调优以及内存管理的信息。

第 7 章 编写存储过程，教会读者与服务器端代码相关的更高级的主题。本章涵盖最重要的服务器端编程语言及其他重要的方面。

第 8 章 管理 PostgreSQL 安全性，本章的目的是帮助读者提高服务器的安全性。本章会介绍从用户管理到行级安全性的各种特性。有关加密的内容也包括在本章中。

第 9 章 处理备份和恢复，涵盖有关备份和数据恢复的所有内容。读者将学到备份其数据以及在遇到灾难时恢复数据。

第 10 章 理解备份和复制，本章与冗余有关。读者将学到异步以及同步复制 PostgreSQL 数据库系统。本章将尽可能全面地介绍所有的现代特性。

第 11 章 选定有用的扩展，本章描述对 PostgreSQL 增加额外功能的被广泛使用的模

块。读者将学到最常见的扩展。

第 12 章 在 PostgreSQL 中排查错误，本章提供了一种系统的方法以修复 PostgreSQL 中的问题。它将使读者能够定位常见的问题并且以一种有条理的方式解决问题。

第 13 章 迁移到 PostgreSQL，它是本书的最后一章并且向读者展示了从商业数据库到 PostgreSQL 的路径。本章将涵盖当今能被迁移的最重要的数据库。

需要的预备知识

本书的读者很广泛。为了能跟得上本书中给出的例子，至少要有一些 SQL 甚至 PostgreSQL 的经验（不过这并非硬性要求）。一般来说，如果能熟悉 Unix 命令行会更好。

适合人群

本书是为那些想要对 PostgreSQL 了解更多并且不满足于基本知识的人而写。其目标是写一本更加深入的书，并且以一种清晰且易懂的方式解释最重要的内容。

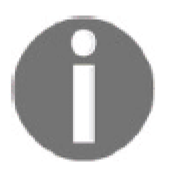
本书约定

在本书中，读者将会找到几种区分不同类别信息的文本样式。这里有这些样式的一些例子及其含义的解释。

任何命令行输入或者输出都被写为下面这样：

```
test=# CREATE TABLE t_test(id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test(name) SELECT 'hans'
FROM generate_series(1, 2000000);
```

新术语和**重要的词**被加粗。



警告或重要的注记出现在一个这样的框中。



提示和技巧以这种形式出现。

读者反馈

我们非常欢迎来自读者的反馈。请让我们知道您对本书的想法——不管您喜欢还是不喜欢这本书。读者反馈对我们来说非常重要，它能帮助我们开发对读者真正有用的主题。

如果要向我们发送一般的反馈，请写邮件到 feedback@packtpub.com，并且在邮件的主题中提及本书的标题。

如果您在一个主题上拥有专业的知识，并且有兴趣写作或者为著书做出贡献，请参考我们的作者指南：www.packtpub.com/authors。

客户支持

现在您已经自豪地拥有了一本由 Packt 出版的书，我们有很多措施来帮助您最大限度地从本次购买中受益。

勘误表

尽管我们已经非常细心地确保内容的准确性，但错误仍可能出现。如果您在我们的书籍中找到了错误，有可能是文字或者代码的错误，请您将错误报告给我们，我们将不胜感激。这样做可以让其他读者免受错误的干扰并且能帮助我们改进本书的后续版本。如果您找到任何勘误，请通过访问 http://www.packtpub.com/submit-errata 并报告：选择相关的书，单击 Errata Submission Form 超链接，然后输入勘误的详情。一旦您的勘误被确认，您的提交将被接受并且那些勘误将被上传至我们的网站或者被加入到该书的勘误表中。

如果要查看之前提交的勘误表，可以访问 https://www.packtpub.com/books/content/support 并且在搜索区中输入该书的名字。要检索的信息将出现在 Errata 部分。

盗版

互联网上对受版权保护的材料被盗版行为是所有媒体都面临着的问题。在 Packt，我们非常重视对我们的版权和许可证的保护。如果您在互联网上发现任何形式的对我们作品的非法复制，请立即向我们提供位置地址或者网站名称，这样我们可以对其进行纠正。

请通过 copyright@packtpub.com 联系我们并提供疑似盗版材料的链接。

感谢您在保护我们的作者和为读者提供有价值内容的能力方面提供的帮助。

问题

如果读者对本书的任何方面有疑问，可以通过 questions@packtpub.com 联系我们，我们将尽力为读者解决。

目　　录

第 1 章　PostgreSQL 概述

PostgreSQL 是世界上最先进的开源数据库系统之一，它拥有很多被开发者和系统管理员广泛使用的特性。在本书中将介绍其中的酷炫特性并且讨论它们的细节。

在本章中，笔者将介绍 PostgreSQL 以及其 9.6 之后版本中的新特性，所有相关的新功能都将被详细地介绍。鉴于 PostgreSQL 代码的变化量巨大且整个项目也很庞大，这里介绍的特性列表当然没法做到面面俱到，因此笔者将尝试把重心放在对大多数人都是最重要的特性上。

本章介绍的特性将被分为以下几类：

- 数据库管理。
- SQL 和开发相关。
- 备份、恢复和复制。
- 性能相关的主题。

1.1　PostgreSQL 9.6 中有什么新技术

PostgreSQL 9.6 发布于 2016 年年末，它是遵循 PostgreSQL 旧的版本编号方式的最后一个版本，这种版本编号方式已经使用了十几年。从 PostgreSQL 10.0 开始将会采用一种新的版本编号系统。从 10.0 开始，主发行将会变得更加频繁。

1.1.1　理解新的数据库管理功能

PostgreSQL 9.6 有很多新特性能够帮助管理员减轻工作负担并且让系统更加鲁棒。

其中之一是 idle_in_transaction_session_timeout 功能。

1.　杀掉闲置会话

在 PostgreSQL 中，一个会话或者事务基本上可以近乎永远生存。在某些情况下，事务存在太久会导致问题。通常，这种情况都是由于缺陷而产生的。其问题在于：不正常的长事务会导致清理出现问题并且可能会发生表膨胀。失控的表增长（膨胀）自然就会导致性能问题，进而引起最终用户的不快。

从 PostgreSQL 9.6 开始，可以限制一个无所事事的事务内部所耗费的数据库连接时

间。例如：

```
test=# SET idle_in_transaction_session_timeout TO 2500;
SET
test=# BEGIN;
BEGIN
test=# SELECT 1;
 ?column?
----------
        1
(1 row)
test=# SELECT 1;
FATAL: terminating connection due to idle-in-transaction timeout
```

管理员和开发者可以设置一个超时时间，在笔者的例子中设置为 2.5 秒。只要一个事务空闲过久，服务器会自动中止该连接。这样，长空闲事务带来的令人不快的副作用就可以通过调整这个参数轻松解决。

2. 在 pg_stat_activity 找到更详尽的信息

pg_stat_activity 功能是一个已经存在了很多年的系统视图，它主要包含了一个活动连接的列表。在老版本的 PostgreSQL 中，管理员能够看到一个查询正在等待其他人，但是没办法查出这个查询为什么等待以及在等待谁。在 9.6 中这种状况得到了改变，pg_stat_activity 中增加了两个列：

```
test=# \d pg_stat_activity
          View "pg_catalog.pg_stat_activity"
       Column      |           Type           | Modifiers
------------------+--------------------------+-----------
 ...
 wait_event_type  | text                     |
 wait_event       | text                     |
...
```

除这种扩充外，9.6 中还增加了一个新的过程，它可以显示谁导致了谁等待：

```
test=# SELECT * FROM pg_blocking_pids(4711);
pg_blocking_pids
------------------
{3435}
(1 row)
```

当这个过程被调用时，它会返回一个导致指定进程阻塞的 PID 列表。

3. 跟踪 vacuum 进度

很多年来，人们都在要求增加一个用于 vacuum 的进度跟踪器。最终，PostgreSQL 9.6 让这一愿望成真，它引入了一个新的系统视图。它可以这样使用：

```
postgres=# SELECT * FROM pg_stat_progress_vacuum ;

  [ RECORD 1 ]      +

pid                 | 29546
datid               | 67535
datname             | test
relid               | 16402
phase               | scanning heap
heap blks total     | 6827
heap blks scanned   | 77
heap blks vacuumed  | 0
index vacuum count  | 0
max_dead_tuples     | 154
num_dead_tuples     | 0
```

PostgreSQL 将会提供有关正在进行的 vacuum 进程的详细信息，这样人们就可以跟踪这一重要操作的进度。

4. 提升 vacuum 速度

PostgreSQL 9.6 并非只为用户提供了更深入的视角来观察 vacuum 当前正在干什么，它还从总体上提升了该处理的速度。从 PostgreSQL 9.6 开始，PostgreSQL 将跟踪所有已冻结页面并且避免清理这类页面。

近乎于只读的表将大大受益于这个改变，因为清理的开销会被急剧降低。

1.1.2 探究新的 SQL 和开发者相关的功能

PostgreSQL 最有前途的新特性之一是执行短语搜索的能力。直到 9.5 都只能搜索词，短语的搜索很难做。9.6 漂亮地解除了这一限制。下面是短语搜索的例子：

```
test=# SELECT phraseto_tsquery('Under pressure') @@
to tsvector('Something was under some sort of pressure');

?column?
```

```
----------

f

(1 row)

test=# SELECT phraseto tsquery('Under pressure') @@
to_tsvector('Under pressure by David Bowie hit number 1 again');

?column?

----------

t

(1 row)
```

第一个查询返回假，因为我们要查找的词没有按照想要的顺序出现。在第二个查询中会返回真，因为确实有一个正确的匹配。

不过，这一特性还不止如此：在 9.6 中可以检查词是否以特定的顺序出现。在下面的例子中，我们想要一个词位于 united 和 nations 之间：

```
test=# SELECT tsquery('united <2> nations') @@
to tsvector('are we really united, happy nations?');

?column?

----------

t

(1 row)

test=# SELECT tsquery('united <2> nations') @@
to tsvector('are we really at united nations?');

?column?

----------

f
```

```
(1 row)
```

第二个查询返回假，因为在 united 和 nations 之间没有词。

1.1.3　使用新的备份和复制功能

PostgreSQL 9.6 也在备份和恢复领域做出了改进。

1. 合理组织 wal_level 和监控

wal_level 设置总是让很多人难以理解。很多人都挣扎于 archive 和 hot_standby 设置之间的区别。为了一并消除这种困惑，两种设置都被替换成了更容易理解的 replica 设置，其效果和 hot_standby 相同。

除此之外，对于复制设置的监控也被简化。在 9.6 以前，只有一个 pg_stat_replication 视图，只能在主服务器上调用它来监督流向从机的数据。现在也可以用 pg_stat_wal_receiver 功能在从机上监控数据流。它基本上就是 pg_stat_replication 功能在从机端的一个镜像，也可以用来判断复制的状态。

2. 使用多个同步后备服务器

PostgreSQL 早就可以执行同步复制了。从 9.6 开始，在 PostgreSQL 中可以有不止一个同步服务器。之前，只有一个服务器需要认可提交。现在可以有一群服务器，它们都必须确认提交。如果用户想要在多节点错误的情况下提升可靠性，这一特性尤其重要。

使用这一新特性的语法很简单：

```
synchronous_standby_names = '3 (server1, server2, server3, server4)
```

不过，PostgreSQL 9.6 中的同步复制还不止如此。之前，PostgreSQL 保证（synchronous_commit = on）事务日志已经到达从机。不过，这并不意味着数据就真正可见。考虑一个例子：某人对主服务器增加了一个用户，马上连接到从机检查该用户。虽然相应的事务日志被保证已经到了从机上，但是不一定能保证该日志中的数据已经对最终用户可见（由于复制冲突等原因）。通过设置 synchronous_commit = 'remote_apply'，现在可以在主服务器的提交之后直接查询从机，而不需要担心数据可能还不可见的问题。remote_apply 值会导致系统比使用 on 值更慢，但是它可用来编写更加高级的应用。

1.1.4　理解性能相关的特性

就像 PostgreSQL 的历次版本发行一样，9.6 中也有很多性能改进，它们都可以帮助应

用提高速度。在本节中，笔者想着重于最重要且最强大的改进。当然，还有很多小的改进没法列举在这里。

1. 改进关系扩展

很多年以来，PostgreSQL 都是逐块地扩展表（或者索引）。在只有一个写入进程的情况下，这还挺不错。但是，在高并发写的情况下，一次写一块就导致竞争且性能不佳。从 9.6 开始，PostgreSQL 开始以一次多块的方式扩展表。一次增加的块数是正在等待的进程数量的 20 倍。

2. 检查点排序和内核交互

当 PostgreSQL 为了检查点把更改写入到磁盘时，现在它会以一种方式来确保这些写入比以往更加有顺序。其根本思路就是在把块写出之前先对它们排序。在这种方式下，随机写将被大幅度抑制，这就会在大部分硬件上得到更高的吞吐量。

排序的检查点并不是 9.6 中唯一的扩展性工作，还有一些新的内核回写配置选项：这有什么意义？在大缓存的情况下，可能需要相当长的一段时间将所有的更改写出。这对具有数百吉字节内存的系统来说尤其令人不快，因为这样会发生相当剧烈的 I/O 风暴。当然，Linux 操作系统层行为可以使用/proc/sys/vm/dirty_background_ratio 命令更改。但是，只有屈指可数的专家和系统管理员真正知道怎么做以及为什么这么做。现在可以使用 checkpoint_flush_after、bgwriter_flush_after 和 backend_flush_after 功能控制刷写行为，通常的规则是尽早进行刷写。作为一种新特性，人们还在积累有关如何更有效使用这些设置的经验。

3. 使用更多先进的外部数据包装器

外部数据包装器已经存在了很多年。从 PostgreSQL 9.6 开始，优化器将会以更加高效的方式使用外部表。其中包括连接下推（连接现在可以在远程执行）以及排序下推（排序现在也能在远程执行）。正因为更快的远程操作，现在在一个集群内分布数据会更加高效。

4. 引入并行查询

传统上，一个查询必须运行在一个 CPU 上。虽然在 OLTP 环境下这种模式还过得去，但对分析型应用来说就有问题，它们会受限于单核的速度。PostgreSQL 9.6 引入了并行查询。当然，实现并行查询很困难，因此所需的一些基础设施也已经开发了多年。现在，所有这些基础设施已经可以为最终用户提供并行的顺序扫描。其思想是在顺序扫描时，让很多 CPU 工作在复杂的 WHERE 条件上。版本 9.6 还允许并行聚集和并行连接。

当然，在这方面还有很多工作需要做，但是我们已经看到了一次大的飞跃。

要控制并行度，有两个基本设置：

```
test=# SHOW max_worker_processes;
 max worker processes
----------------------
 8
(1 row)

test=# SHOW max_parallel_workers_per_gather;
 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)
```

第一个限制可用工作者进程的总体数量。第二个控制每一个收集节点允许使用的工作者数量。



在执行计划中用户将看到收集节点这种新事物。它负责统一来自并行子进程的结果。除了这些基础设置之外，还有一些新的优化器参数可以用来调整并行查询的代价。

5. 增加 snapshot too old

用过 Oracle 的人应该都见过下面的错误消息：snapshot too old。在 Oracle 中，这种消息表示一个事务已经持续太久，因此它必须被中断。在 PostgreSQL 中，事务几乎可以无限期运行。不过，长事务仍然可能成为问题，因此 9.6 中增加了 snapshot too old 的新特性，它允许一定时间之后中止事务。

这个特性的思想是防止表膨胀并且让最终用户认识到他们可能做了傻事。

1.2 总　　结

PostgreSQL 9.6 和 10.0 增加了很多功能，它们允许人们运行更加专业的应用，并且能运行得更快、更高效。至于 PostgreSQL 10.0，准确的新特性还没有被完全定义出来，不过本章中已经介绍了一些已知的内容。

第 2 章　理解事务和锁定

锁定在任何一种数据库中都是一个重要的主题。仅仅理解它如何帮助写出更正确或者更好的应用程序是不够的，更重要的是从性能的角度来理解它。如果没有正确地处理锁，用户的应用程序可能不止是速度慢，它还可能出现错误并且做出愚蠢的行为。在笔者看来，锁定是性能的关键，对它有一个好的概述无疑会有帮助。因此，理解锁定和事务对于管理员和开发者同样重要。

在本章中，读者将学到：

- 基本锁定。
- 事务和事务隔离。
- 死锁。
- 锁定和外键。
- 显式和隐式锁定。
- 咨询锁。

在本章的最后，读者将学会以最有效的方式理解和利用 PostgreSQL 的事务。

2.1　使用 PostgreSQL 事务

PostgreSQL 提供了一种非常先进的事务机制，它为开发者和管理员提供了无数的特性。在本节中，将先看看基本的概念。

要知道的第一个要点是：在 PostgreSQL 中，每一个操作都是一个事务。如果用户向服务器发送一个简单查询，它就已经是一个事务。这里有一个例子：

```
test=# SELECT now(), now();
              now              |              now
-------------------------------+-------------------------------
 2016-08-30 12:03:27.84596+02 | 2016-08-30 12:03:27.84596+02
(1 row)
```

在这种情况下，SELECT 语句将是一个单独的事务。如果再次执行同一个语句，将返回一个不同的时间戳。



记住 now 函数将会返回该事务的时间。因此，SELECT 语句总是返回两个相同的时间戳。

如果要让多个语句作为同一个事务的一部分，就必须使用 BEGIN 子句：

```
test=# \h BEGIN
Command:     BEGIN
Description: start a transaction block
Syntax:
BEGIN [ WORK | TRANSACTION ] [ transaction mode [, ...] ]

where transaction mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ
   | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
   [ NOT ] DEFERRABLE
```

BEGIN 子句将确保多个命令被包装到一个事务中。下面的例子展示了它的作用：

```
test=# BEGIN;
BEGIN
test=# SELECT now();
              now
-------------------------------
 2016-08-30 12:13:54.839277+02
(1 row)
test=# SELECT now();
              now
-------------------------------
 2016-08-30 12:13:54.839277+02
(1 row)
test=# COMMIT;
COMMIT
```

这里的重点是两个时间戳将是完全一样的。正如之前提到的，我们这里所说的是事务的时间。

要结束事务，可以使用 COMMIT：

```
test=# h COMMIT
Command:     COMMIT
Description: commit the current transaction
Syntax:
COMMIT [ WORK | TRANSACTION ]
```

这里用到了几个语法元素，用户可以用 COMMIT、COMMIT WORK 或者 COMMIT

TRANSACTION。所有这些选项都具有相同的含义。如果这还不够，还有更多的选择：

```
test=# h END
Command:     END
Description: commit the current transaction
Syntax:
END [ WORK | TRANSACTION ]
```

END 子句和 COMMIT 子句相同。

ROLLBACK 是和 COMMIT 对应的命令，但它并不是成功地结束一个事务，它仅会停止事务而不把事务中的部分对其他事务可见：

```
test=# h ROLLBACK
Command:     ROLLBACK
Description: abort the current transaction
Syntax:
ROLLBACK [ WORK | TRANSACTION ]
```

有些应用会使用 ABORT 来替代 ROLLBACK，其含义完全相同。

2.1.1　在事务内处理错误

事务并不总是从头到尾都正确。不过，在 PostgreSQL 中，只有没有错误的事务才能被提交。这里可以看到提交出错的事务时会发生什么：

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
 ?column?
----------
        1
(1 row)

test=# SELECT 1 / 0;
ERROR: division by zero
test=# SELECT 1;
ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=# COMMIT;
ROLLBACK
```

注意除零是算不出来的。



在任何正确的数据库中，像这样的一条指令将立刻出现错误并且让该语句失败。

重点要指出的是，和 MySQL 不一样，PostgreSQL 将会出错，而前者看起来没有算术错误结果的问题。

在一个错误出现后，即便后面的指令在语义和语法上都是正确的，也将不会再有语句被接受。这种情况下，仍然可以发出一个 COMMIT。不过，PostgreSQL 将回滚该事务，因为在这种状况下能做的事情也就只有回滚了。

2.1.2　使用保存点

在专业应用中，很难写出长度合理的事务而且在运行中永不出错。为了解决这一问题，用户可以利用所谓的 SAVEPOINT。顾名思义，它是事务中的一个安全位置，在出现错误事件时应用可以返回到这些位置。这里有一个例子：

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
 ?column?
----------
        1
(1 row)

test=# SAVEPOINT a;
SAVEPOINT
test=# SELECT 2 / 0;
ERROR: division by zero
test=# ROLLBACK TO SAVEPOINT a;
ROLLBACK
test=# SELECT 3;
 ?column?
----------
        3
(1 row)

test=# COMMIT;
COMMIT
```

在第一个 SELECT 子句后面，笔者决定创建一个 SAVEPOINT 来确保应用总是可以在事务内返回到这一点。如你所见，SAVEPOINT 有一个名字，在后面会用这个名字来引

用它。

在返回到 a 之后，事务可以正常地继续下去。代码已经跳回到错误发生之前，所以所有事情都是正常的。

一个事务内部的保存点数量没有限制。我们已经见过有客户在一个单一操作中使用超过 250000 个保存点。PostgreSQL 可以轻松地处理这样的情况。

如果用户想在一个事务内移除一个保存点，有一个命令 RELEASE SAVEPOINT 可用：

```
test=# h RELEASE SAVEPOINT
Command:     RELEASE SAVEPOINT
Description: destroy a previously defined savepoint
Syntax:
RELEASE [ SAVEPOINT ] savepoint_name
```

很多人会问，如果在一个事务结束之后尝试到达一个保存点会发生什么？答案是，事务结束时保存点的寿命也随之终止。换句话说，在事务已经被结束之后无法返回到一个特定的点。

2.1.3　事务性 DDL

很不幸，PostgreSQL 有一些很好的特性在很多商业数据库系统中并不存在。在 PostgreSQL 中，可以在一个事务块中运行 DDL（改变数据结构的命令）。在一个典型的商业系统中，在当前事务中的 DDL 将会被隐式地提交，但在 PostgreSQL 中不是这样。除去少量例外（DROP DATABASE、CREATE TABLESPACE/DROP TABLESPACE 等），PostgreSQL 中所有的 DDL 都是事务性的，这会给最终用户带来巨大的好处和真正的实惠。

这里有一个例子：

```
test=# d
No relations found.
test=# BEGIN;
BEGIN
test=# CREATE TABLE t_test(id int);
CREATE TABLE
test=# ALTER TABLE t_test ALTER COLUMN id TYPE int8;
ALTER TABLE
test=# d t_test
    Table "public.t_test"
```

```
 Column |  Type  | Modifiers
--------+--------+-----------
 id     | bigint |

test=# ROLLBACK;
ROLLBACK
test=# d t_test
Did not find any relation named "t_test".
```

在这个例子中创建并修改了一个表，然后整个事务被立刻中止。如你所见，其中没有隐式的 COMMIT 或者任何其他奇怪的行为，PostgreSQL 也能按照我们的期望工作。

如果用户想要部署软件，事务性 DDL 特别重要。想象运行一个 CMS 的场景。如果发布了一个新的版本，用户将会想要升级。单独运行旧版本或者新版本可能都是可以的，但是用户不希望得到新旧版本的混合体。因此，将升级放在一个单一事务内无疑会大有益处，因为这会使升级变成一个原子操作。



psql 允许用户使用 i 指令包括文件。它允许用户开始一个事务、载入一些文件并且在一个单一事务中执行它们。

2.2　理解基本的锁定

在本节中，读者将学到基本的锁定机制。本节的目标是让读者理解锁通常是如何工作的以及如何让简单应用正确运行。作为例子，可以创建一个简单表。为了便于展示，笔者将增加一行到该表中：

```
test=# CREATE TABLE t test (id int);
CREATE TABLE
test=# INSERT INTO t test VALUES (1);
INSERT 0 1
```

重点是表可以被并发读取。很多同时读取相同数据的用户不会彼此阻塞。这允许 PostgreSQL 处理上千个用户而不出问题。



多个用户可以同时读取同样的数据而不彼此阻塞。

现在的问题是，如果读写同时产生会发生什么？这里有一个例子，如表 2-1 所示。

表 2-1

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
用户将看到 1	SELECT * FROM t_test;
	用户将看到 1
COMMIT;	COMMIT;

两个事务被打开。第一个将更改一行。不过这不会有问题，因为第二个事务可以继续。它将返回旧的行，也就是在 UPDATE 之前的行。这种行为被称为**多版本并发控制（MVCC）**。

注意事务将只能看到已经被写事务提交的数据。一个事务不能观察到由一个活跃连接造成的改变。



一个事务只能看到已经被提交的更改。

还有第二个重要的方面：（截至 2017 年）很多商业或者开源数据库仍然不能处理并发读写。在 PostgreSQL 中绝对没有这样的问题，读写是可以共存的，写事务不会阻塞读事务。在数据被提交之后，该表将会包含 2。

如果两个用户同时修改同一数据会发生什么？这里有个例子，如表 2-2 所示。

表 2-2

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
它将返回 3	UPDATE t_test SET id = id + 1 RETURNING *;
	它将等待事务 1
COMMIT;	它将等待事务 1
	它现在将重新读取，并且找到 3、设置值，然后返回 4
	COMMIT;

假设用户想要统计一个网站的点击数。如果运行刚才概述的代码，不会有点击会被漏掉，因为 PostgreSQL 保证一个 UPDATE 会在其他更新之后被运行。



PostgreSQL 只会锁定被 UPDATE 影响的行。因此如果有 1000 行，理论上用户可以在同一个表上运行 1000 个并发更改。

还值得提到的一点是，用户总是可以运行并发读。我们的两个写操作将不会阻塞读。

- 避免典型错误和显式锁定

在笔者作为一个职业 PostgreSQL 顾问（http://postgresql-support.de/）的生涯中，已经见过了不少被一次又一次重复的错误。如果生命中有什么是永恒的，那么这些典型错误绝对是一些从不改变的。

下面是笔者的最爱，如表 2-3 所示。

表 2-3

事务 1	事务 2
BEGIN;	BEGIN;
SELECT max(id) FROM product;	SELECT max(id) FROM product;
用户将看到 17	用户将看到 17
用户将决定用 18	用户将决定用 18
INSERT INTO product ... VALUES (18, ...)	INSERT INTO product ... VALUES (18, ...)
COMMIT;	COMMIT;

在这一情况中，将会有一次重复键违规或者两个完全一样的项。这个问题的两个变种都不是我们想要的。

修正这个问题的一种方式是使用显式表锁定：

```
test=# h LOCK
Command:     LOCK
Description: lock a table
Syntax:
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]

where lockmode is one of:

   ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE |
   SHARE UPDATE EXCLUSIVE| SHARE |
   SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

如你所见，PostgreSQL 为锁定一个表提供了 8 种类型的锁。在 PostgreSQL 中，一个锁可以轻如 ACCESS SHARE 锁或者重如 ACCESS EXCLUSIVE 锁。下面展示了这些锁会

做什么。

- ACCESS SHARE：这种类型的锁由读操作获取，它只与 ACCESS EXCLUSIVE 冲突，后者会由 DROP TABLE 之类的操作获取。事实上，这意味着如果一个表就要被删除，则不能开始一个 SELECT。这也暗示着 DROP TABLE 不得不等待直到一个读取事务完成。
- ROW SHARE：PostgreSQL 会在 SELECT FOR UPDATE/SELECT FOR SHARE 的情况下取得这种锁。它与 EXCLUSIVE 以及 ACCESS EXCLUSIVE 冲突。
- ROW EXCLUSIVE：这种锁由 INSERT、UPDATE 和 DELETE 取得。它与 SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 还有 ACCESS EXCLUSIVE 冲突。
- SHARE UPDATE EXLUSIVE：这种锁由 CREATE INDEX CONCURRENTLY、ANALYZE、ALTER TABLE、VALIDATE 和其他和 VACUUM（非 VACUUM FULL）一样的 ALTER TABLE 形式取得。它与 SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE 以及 ACCESS EXCLUSIVE 锁模式冲突。
- SHARE：当一个索引被创建时，将会设置 SHARE 锁。它与 ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE 和 ACCESS EXCLUSIVE 冲突。
- SHARE ROW EXCLUSIVE：这种锁由 CREATE TRIGGER 和某些形式的 ALTER TABLE 设置，并且与除了 ACCESS SHARE 之外的所有模式都冲突。
- EXCLUSIVE：这种类型的锁是目前为止最严格的一种。它保护操作不受读和写的影响。如果一个事务取得了这种锁，任何其他人都不能读取或者写入受影响的表。

有了这样的 PostgreSQL 锁定结构，对于之前的最大值问题的一种解决方案可以是：

```
BEGIN;
LOCK TABLE product IN ACCESS EXCLUSIVE MODE;
INSERT INTO product SELECT max(id) + 1, ... FROM product;
COMMIT;
```

要记住对于此类操作这是一种很不好的方式，因为在操作期间没有任何其他人可以读取或者写入该表。因此，应该不惜一切代价避免 ACCESS EXCLUSIVE。

- 考虑可替换的解决方案

不过，还是有一种替代解决方案可以用于该问题。考虑下面的例子：税务所要求你

编写一个生成发票编号的应用。税务所可能要求你创建的发票编号没有间隔也没有重复。你应该怎么做呢？当然，一种方案会是表锁。但是你实际上可以做得更好。下面看看笔者会怎么做：

```
test=# CREATE TABLE t_invoice (id int PRIMARY KEY);
CREATE TABLE
test=# CREATE TABLE t_watermark (id int);
CREATE TABLE
test=# INSERT INTO t watermark VALUES (0);
INSERT 0 1
test=# WITH x AS (UPDATE t_watermark SET id = id + 1 RETURNING *)
         INSERT INTO t_invoice
         SELECT * FROM x RETURNING *;
 id
----
  1
(1 row)
```

在这种情况下，笔者引入了一个名为 t_watermark 的表，它只包含一行。WITH 部分被首先执行。该行将被锁定并且增加，得到的新值将被返回。一个时间点只有一个人能这样做。接着，CTE 返回的值被用在发票表中，它被保证是唯一的。其中最美妙的事情是只在 watermark 表上有一个简单的行锁，在发票表上的读取操作将不会被阻塞。总的来说，这种方法更具有可扩展性。

2.3　使用 FOR SHARE 和 FOR UPDATE

有时，应用会从数据库中选择一些数据，然后对它们做一些处理并且最终将一些被更改的数据存回到数据库，这是 SELECT FOR UPDATE 的一种典型用例。

下面是一个例子：

```
BEGIN;
SELECT * FROM invoice WHERE processed = false;
** application magic will happen here **
UPDATE invoice SET processed = true ...
COMMIT;
```

这里的问题是两个人可能选择相同的未处理数据。那么对于那些已处理行的更改将

会被覆盖。简而言之，将会发生一种竞争的情况。

为了解决这一问题，开发者可以使用 SELECT FOR UPDATE。例如：

```
BEGIN;
SELECT * FROM invoice WHERE processed = false FOR UPDATE;
** application magic will happen here **
UPDATE invoice SET processed = true ...
COMMIT;
```

SELECT FOR UPDATE 将会像 UPDATE 那样锁住行，这意味着不会发生并发的更改。所有的锁会照常在提交时被释放。如果一个 SELECT FOR UPDATE 在等待其他某个 SELECT FOR UPDATE，那前者就不得不等到后者结束（COMMIT 或者 ROLLBACK）。如果被等待的事务不想结束，那么等待着的事务可能会永远等待。

为了避免这种情况，可以使用 SELECT FOR UPDATE NOWAIT，如表 2-4 所示。

可以这样做：

表 2-4

事务 1	事务 2
BEGIN;	BEGIN;
SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;	
一些处理	SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;
一些处理	ERROR: could not obtain lock on row in relation tab

如果用户觉得 NOWAIT 不够灵活，可以考虑使用 lock_timeout，它表示用户想要花在等待锁上的时间。可以在会话级别上设置这个参数：

```
test=# SET lock_timeout TO 5000;
SET
```

在上例中，该值被设为 5 秒。

虽然 SELECT 基本不做锁定，但 SELECT FOR UPDATE 可能会相当严格。想象一下下面的业务处理：我们想填满一架提供 200 个座位的飞机，很多人想要并发地预订座位。在这种情况下，将会发生表 2-5 所示的情景。

表 2-5

事务 1	事务 2
BEGIN;	BEGIN;
SELECT ... FROM flight LIMIT 1 FOR UPDATE;	
等待用户输入	SELECT ... FROM flight LIMIT 1 FOR UPDATE;
等待用户输入	必须等待

这里的问题在于在一个时间点上只能有一个座位被预订。可以用来预订的座位实际上有 200 个，但是每一个人都必须等待第一个人。虽然第一个座位已经被阻塞，但是即便人们并不在乎最后得到哪个座位，也没有其他人可以预订其他的座位。

SELECT FOR UPDATE SKIP LOCKED 将会修复这一问题。让我们先创建一些示例数据：

```
test=# CREATE TABLE t_flight AS
   SELECT * FROM generate_series(1, 200) AS id;
SELECT 200
```

下面是见证奇迹的时刻，如表 2-6 所示。

表 2-6

事务 1	事务 2
BEGIN;	BEGIN;
SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;	SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
会返回 1，2	会返回 3，4

如果每个人都想要取得两行，可以在同一时间服务 100 个并发事务而不需要担心阻塞事务。



要记住，等待是最慢的一种执行形式。如果在一个时间点只有一个 CPU 可以活动，购买更大的服务器就没有意义。

不过还有更多需要考虑的事情。在某些情况下，FOR UPDATE 可能有意想不到的结果。大部分人并没有意识到 FOR UPDATE 将会影响外键这一事实。让我们假定有两个表：一个存储货币，另一个存储账户：

```
CREATE TABLE t currency (id int, name text, PRIMARY KEY (id));
INSERT INTO t_currency VALUES (1, 'EUR');
```

```
INSERT INTO t_currency VALUES (2, 'USD');

CREATE TABLE t_account (id int, currency_id int REFERENCES t_currency (id)
ON UPDATE CASCADE ON DELETE CASCADE, balance numeric);
INSERT INTO t_account VALUES (1, 1, 100);
INSERT INTO t_account VALUES (2, 1, 200);
```

现在想要在账户表上运行 SELECT FOR UPDATE，如表 2-7 所示。

表 2-7

事务 1	事务 2
BEGIN;	
SELECT * FROM t_account FOR UPDATE;	BEGIN;
等待用户继续	UPDATE t_currency SET id = id * 10;
等待用户继续	它将等待事务 1

尽管是在账户表上有 SELECT FOR UPDATE，但货币表上的 UPDATE 却将阻塞。这样做是有必要的，否则就会有机会破坏外键约束。因此，在比较复杂的数据结构中，可以很容易地避免在最不想出现竞争的地方（一些非常重要的查阅表）碰到竞争。

在 FOR UPDATE 之上，还有 FOR SHARE、FOR NO KEY UPDATE 以及 FOR KEY SHARE。下面的列表描述了这些模式究竟意味着什么。

- FOR NO KEY UPDATE：这种模式很像 FOR UPDATE。不过，这种锁要更弱一些并且因此能和 SELECT FOR SHARE 共存。
- FOR SHARE：FOR UPDATE 是一种相当强的锁，它假设用户肯定会去更改行。FOR SHARE 与之不同，因为可以有多于一个事务在同一时刻都持有 FOR SHARE 锁。
- FOR KEY SHARE：这种模式的行为类似于 FOR SHARE，不过这种锁较弱。它将阻塞 FOR UPDATE 但不会阻塞 FOR NO KEY UPDATE。

要理解这些锁模式的含义，最简单直观的方式就是进行尝试并且观察会发生什么。改进锁定行为确实非常重要，因为它能大幅度地提升应用的可扩展性。

2.4　理解事务隔离级别

到目前为止，读者已经见到了如何处理锁定以及一些基本的并发。在本节中，读者将学到事务隔离。对笔者而言，这是现代软件开发中最容易被忽略的主题之一。只有一小部分软件开发者真正意识到了这个问题，而它又会导致令人反感而且难以置信的

缺陷。

下面是有关于此的一个例子，如表 2-8 所示。

表 2-8

事务 1	事务 2
BEGIN;	
SELECT sum(balance) FROM t_account;	
用户将看到 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	
用户将看到 400	
COMMIT;	

不管第二个事务怎样，大部分用户实际上会期望左边的事务一直返回 300，但是事实并非如此。默认情况下，PostgreSQL 运行在 READ COMMITTED 事务隔离模式中。这意味着事务中的每个语句都将得到数据的一个新快照，它在整个查询期间都不变。



一个 SQL 语句将在同一个快照上进行操作并且会忽略在它运行期间并发事务所作的更改。

如果用户想避免这种情况，可以使用 TRANSACTION ISOLATION LEVEL REPEATABLE READ。在这种事务隔离级别中，一个事务将在整个事务期间都使用同一个快照。下面是使用这种隔离级别的例子，如表 2-9 所示。

表 2-9

事务 1	事务 2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
SELECT sum(balance) FROM t_account;	
用户将看到 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	SELECT sum(balance) FROM t_account;
用户将看到 300	用户将看到 400
COMMIT;	

如上所示，第一个事务将会冻结它的数据快照，并且在整个事务期间都为我们提供一致的结果。如果用户想要运行报表，这个特性将会特别重要。一份报表的第一页和最后一页应该总是一致的，并且都在相同的数据上操作。因此，可重复读是让报表一致的关键。

注意隔离相关的错误不会总是立刻跳出来，它可能会在一个应用移到生产环境后数年才被发现。



可重复读并不比读已提交更昂贵，所以不需要担心性能上受到的惩罚。

- 考虑 SSI 事务

在读已提交和可重复读之上，PostgreSQL 提供了可序列化（简称 SSI）事务。因此，PostgreSQL 总共支持 3 种隔离级别。注意读未提交（在一些商业数据库中还是作为默认级别）不受支持：如果用户尝试开始一个读未提交的事务，PostgreSQL 会悄无声息地把它映射成读已提交。不过，现在让我们还是回到可序列化。

可序列化背后的思想很简单，如果在只有一个用户的情况下一个事务是正确工作的，在选择这种隔离级别时它也能在并发情况下工作。不过，用户必须做好准备：事务可能会失败（设计就是这样）并且报错。除此之外，还会付出性能上的代价。

如果读者想要了解更多有关这种隔离级别的内容，可以考虑看看 https://wiki.postgresql.org/wiki/Serializable。



只有当用户对数据库引擎内部有深入的了解时才考虑使用可序列化。

2.5　观察死锁和类似的问题

死锁是一个重要的问题，它可能会发生在每一个笔者知道的数据库中。大体上，如果两个事务不得不相互等待，就将发生一次死锁。

在本节中，读者将看到怎么会发生死锁。让我们假设有一个包含两行的表：

```
CREATE TABLE t_deadlock (id int);
INSERT INTO t_deadlock VALUES (1), (2);
```

表 2-10 展示了会发生什么：

表 2-10

事务 1	事务 2
BEGIN;	BEGIN;
UPDATE t_deadlock SET id = id * 10 WHERE id = 1;	UPDATE t_deadlock SET id = id * 10 WHERE id = 2;
UPDATE t_deadlock SET id = id * 10 WHERE id = 2;	
等待事务 2	UPDATE t_deadlock SET id = id * 10 WHERE id = 1;
等待事务 2	等待事务 1
	1 秒（deadlock_timeout）以后死锁会被解除
COMMIT;	ROLLBACK;

一旦检测到死锁，将会显示下面的错误消息：

```
ERROR: deadlock detected
DETAIL: Process 91521 waits for ShareLock on transaction 903; blocked by
process 77185.
Process 77185 waits for ShareLock on transaction 905; blocked by process
91521.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,1) in relation "t_deadlock"
```

PostgreSQL 甚至会告诉我们哪一行导致了该冲突。在笔者的例子中，所有罪恶的根源是元组(0,1)。这里读者能看到的是一个 ctid，它告诉我们一行在表中的物理位置。在这个例子中，它是第一块（0）中的第一行。

如果这一行对用户的事务还可见，甚至可以查询这一行：

```
test=# SELECT ctid, * FROM t_deadlock WHERE ctid = '(0, 1)';
  ctid | id
-------+----
 (0,1) | 1
(1 row)
```

要记住如果一个行已经被删除或者修改，这个查询是不会返回该行的。

不过，它并非是死锁导致事务失败的唯一情况。当事务由于一些原因没有被序列化时，也可能发生同样的事情。表 2-11 会展示这一现象。为了让这个例子能够工作，笔者假定表中仍然有两行：id=1 和 id=2。

表 2-11

事务 1	事务 2
BEGIN ISOLATION LEVEL REPEATABLE READ;	
SELECT * FROM t_deadlock;	
将返回两行	
	DELETE FROM t_deadlock;
SELECT * FROM t_deadlock;	
将返回两行	
DELETE FROM t_deadlock;	
事务将会出错	
ROLLBACK; -我们不再能 COMMIT	

在这个例子中，两个并发事务在运行。只要事务 1 仅选择数据，一切都不会有问题，因为 PostgreSQL 可以轻易地保存静态数据的幻象。但是如果第二个事务提交了一次 DELETE 会发生什么呢？只要事务 1 中仅执行读，则仍然不会有问题。当事务 1 尝试删除或者修改数据时麻烦就开始了，因为这些数据在这一时刻实际上已经死亡了。对于 PostgreSQL 来说，这里唯一的解决方案就是报错：

```
test=# DELETE FROM t deadlock;
ERROR: could not serialize access due to concurrent update
```

实际上，这意味着最终用户不得不准备好处理出错的事务。如果出了差错，正确编写的应用必须能够进行重试。

2.6　利用咨询锁

PostgreSQL 有一套非常有效而且复杂的事务机制，它能够以非常细粒度和高效的方式来处理锁。数年前，有人冒出了使用这套代码来在应用之间同步的想法。这样，咨询锁便诞生了。

在使用咨询锁时，一定要提及的是它们不会像普通锁那样在 COMMIT 后就消失。因此，确保以正确可靠的方式完成解锁是非常重要的。

如果用户决定使用一个咨询锁，他/她实际锁住的是一个数字。所以这与行或者数据无关：它真的就是一个数字。下面是怎么用咨询锁的例子，如表 2-12 所示。

表 2-12

会话 1	会话 2
BEGIN;	
SELECT pg_advisory_lock(15);	
	SELECT pg_advisory_lock(15);
	它必须等待
COMMIT;	它仍然要等待
SELECT pg_advisory_unlock(15);	它还在等待
	锁被得到

第一个事务将锁住 15。第二个事务不得不等待，直到这个数字被再次解锁。在第一个事务已经提交后，第二个会话还将等待。这是非常重要的，因为我们不能指望事务会完美地结束并且神奇地为我们解决这些事情。

如果用户想要解锁所有锁住的数字，PostgreSQL 提供了 pg_advisory_unlock_all()函数来做这件事：

```
test=# SELECT pg advisory unlock all();
 pg advisory unlock all
-------------------------

(1 row)
```

有时候用户可能想看看是否能得到一个锁并且在不能得到时报错。为了实现这种功能，PostgreSQL 提供了几个函数。如果用户使用 df *try*advisory*，PostgreSQL 会返回所有可用函数的列表。

2.7　优化存储以及控制清理

事务是 PostgreSQL 系统一个完整的部分。不过，伴随事务的还有一点小小的代价。正如本章中已经展示过的，可能会出现为并发用户展现不同数据的情况。对于一个查询，不是每一个人都会得到相同的返回数据。除此之外，DELETE 和 UPDATE 不能直接覆盖数据，因为那样 ROLLBACK 就将无法工作。如果用户恰好处于一次大型 DELETE 操作的中间，他/她将不能确定是否能够 COMMIT。还有，当用户执行 DELETE 时，涉及的数据仍然是可见的，甚至有时在修改完成很久以后数据都是可见的。

所以，这意味着清理不得不异步地发生。事务不能够清除自己留下的垃圾并且在 COMMIT/ROLLBACK 时处理死亡行也为时过早。

这个问题的解决方案就是 VACUUM：

```
test=# h VACUUM
Command:     VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ]
[ table name [(column name [, ...] ) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table_name
[ (column_name[, ...] ) ] ]
```

VACUUM 将访问所有可能包含修改的页面并且找出所有的死亡空间。找到的空闲空间将交由该关系的**空闲空间映射（FSM）**跟踪。

注意在大部分情况下，VACUUM 将不会收缩表的尺寸。相反，它将在现有的存储文件中跟踪并且寻找空闲空间。



在 VACUUM 后，表通常将具有操作前相同的尺寸。如果在一个表的末尾没有合法的行，在极少的情况下文件尺寸会减小，但这不是规则而是一种特例。

这对最终用户的意义将在 2.7.2 节中介绍。

2.7.1　配置 VACUUM 和 autovacuum

回到 PostgreSQL 项目的早期，人们不得不手工运行 VACUUM。幸运地是这已经成为过去式，现如今的管理员们可以依靠一个被称作 autovacuum 的工具，它是 PostgreSQL 服务器基础设施的一部分。autovacuum 会自动地处理清理并且在后台工作，它每一分钟（见 postgresql.conf 中的 autovacuum_naptime = 1）醒来一次并且检查是否有工作要做。如果有工作要做，autovacuum 将派生出最多 3 个（见 postgresql.conf 中的 autovacuum_max_workers）工作者进程。

主要的问题是：什么时候 autovacuum 会触发一个工作者进程的创建？



实际上 autovacuum 进程并不自己派生进程。相反，它会告诉主进程来做这件事。这是为了防止在失败的情况下产生僵尸进程，并且提高鲁棒性。

这个问题的答案还可以在 postgresql.conf 中找到：

```
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
```

autovacuum_vacuum_scale_factor 告诉 PostgreSQL：如果一个表中 20%的数据已经被改过，那么它就值得被清理。麻烦在于，如果一个表只由一行构成，一次更改就已经是 100%。派生出一个完整的进程来只清理一行绝对是没有意义的。因此，autovacuum_vacuuum_ threshold 会要求我们需要 20%的修改并且那 20%必须有至少 50 行。否则，VACUUM 不会开始。在进行优化器统计信息创建时也使用了同样的机制。至少需要 10%的修改并且至少 50 行才能触发对优化器统计信息的更新。在理想情况下，autovacuum 会在一次普通的 VACUUM 中创建新的统计信息以避免对表不必要的访问。

1. 探究事务回卷相关的问题

postgresql.conf 中还有另外两个需要着重理解的设置：

```
autovacuum_freeze_max_age = 200000000
autovacuum_multixact_freeze_max_age = 400000000
```

为了理解整个问题，了解 PostgreSQL 怎样处理并发是很重要的。PostgreSQL 的事务机制建立在对事务 ID 的检查以及事务所处状态的基础之上。

一个例子：如果我是 ID 为 4711 的事务并且你正好是 4712，我将看不到你，因为你还在运行中。如果我是 ID 为 4711 的事务而你是 3900，如果你已经提交，我将能够看到你，而如果你失败，我将忽略你的操作。

接着麻烦就来了：事务 ID 是有限的，不是我们可以任意挥霍的。在某个时候，它们将会开始回卷。这就意味着事务号 5 可能实际上在事务号 800000000 之后。PostgreSQL 是怎么知道哪一个在前？它通过存储一个水位标志来做到这一点。在某个时候，那些水位标志会被调整，并且这正好就是 VACUUM 开始介入的时候。通过运行 VACUUM（或者 autovacuum），用户可以确保水位标志被调整后，总是有足够的未来事务 ID 可用。

不是每一个事务都会增加事务 ID 计数器。只要一个事务还在读取，它将只有一个虚拟事务 ID。这保证了事务 ID 不会增加得太快。

autovacuum_freeze_max_age 定义了在执行一次 VACUUM 操作来阻止表中事务 ID 回卷之前，一个表的 pg_class.relfrozenxid 域能达到的最大事务数（年龄）。这个值相当低，因为它也影响着 clog 清理（clog 或者提交日志是一个为每个事务存储 2 位的数据结构，这 2 位表示事务是运行中、已中止、已提交或是正在子事务中）。

autovacuum_multixact_freeze_max_age 配置在执行一次 VACUUM 操作来阻止表中 multixact ID 回卷之前，一个表的 pg_class.relminmxid 域能达到的最大年龄（以 multixact 事务个数计）。冻结元组是一种重要的性能问题，在第 6 章中会有更多关于它的内容。

2. 关于 VACUUM FULL 的一点建议

除了普通的 VACUUM，还可以使用 VACUUM FULL。不过，有一点笔者必须要强调：VACUUM FULL 实际上会锁住表并且重写整个关系。如果是一个小的表，这可能不成问题。但如果表很大，这种表锁会影响用户数分钟之久！VACUUM FULL 会阻塞即将到来的写操作并且有些人会觉得数据库看上去已经宕掉了。因此，关于这个问题人们已经给出了很多警示。

要消除掉 VACUUM FULL，笔者建议读者去看看 pg_squeeze（http://www.cybertec.at/introducing-pg_squeeze-a-postgresql-extension-to-auto-rebuild-bloated-tables/），它可以重写一个表而不用阻塞写操作。

2.7.2　观察工作中的 VACUUM

在这些介绍之后，是时候看看运转中的 VACUUM 了。之所以在这里包括这一节，是因为笔者作为一个 PostgreSQL 顾问和支持人员（http://postgresql-support.de/）的实践表明，大部分人对存储端发生的事情只有着非常模糊的理解。

下面会再次强调这一点，在大部分情况下 VACUUM 将不会收缩你的表，空间通常不会被交还给文件系统。

这里是笔者的例子：

```
CREATE TABLE t test (id int) WITH (autovacuum enabled = off);

INSERT INTO t test
   SELECT * FROM generate_series(1, 100000);
```

其想法是创建一个包含 100000 行的简单表。注意可以对特定的表关闭 autovacuum，但对于大部分应用这通常不是个好主意。不过，在一些极限情况中 autovacuum_enabled = off 是有意义的。只要考虑一个生命周期很短的表。如果开发者已经知道整个表将在数秒之后被删除，对它清理元组就没有意义了。在数据仓库中，如果用户把表用作暂存区就会是这样。在这个例子中，VACUUM 被关闭以保证不会在后台发生其他事情，读者所看到的都是由笔者触发而不是被某个其他进程所执行。

首先检查该表的尺寸：

```
test=# SELECT pg size pretty(pg relation size('t test'));
 pg size pretty
----------------
 3544 kB
(1 row)
```

pg_relation_size 返回一个表的字节尺寸，而 pg_size_pretty 将得到这个数字，并且把它转换成人类可读的。

然后该表中的所有行都将被更新：

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
```

所发生的事情对理解 PostgreSQL 非常重要，数据库引擎不得不复制所有的行。为什么会这样？首先，我们不知道该事务是否将会成功，因此这些数据不能被覆盖。第二个重要的方面是，可能有一个并发事务仍然在用这些数据的旧版本。



UPDATE 操作将会复制行。

逻辑上，该表的尺寸将会在修改之后变得更大：

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
----------------
 7080 kB
(1 row)
```

在 UPDATE 之后，人们可能会尝试把空间还给文件系统：

```
test=# VACUUM t_test;
VACUUM
```

如前所述，大部分情况下 VACUUM 不把空间还给文件系统。相反，它允许空间被重用。因此，该表根本不会收缩：

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
----------------
 7080 kB
(1 row)
```

不过，下一次 UPDATE 不会让该表长大，因为它会用掉该表中的空闲空间。只有第二次的 UPDATE 会让该表再次长大，因为所有的空间都用完了，需要额外的存储空间：

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
```

```
----------------
 7080 kB
(1 row)

test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
----------------
 10 MB
(1 row)
```

如果笔者必须选择一项读者在读完本书后应该记住的内容，这算一项。理解存储通常是性能和管理的关键。

让我们运行更多的查询：

```
VACUUM t_test;
UPDATE t_test SET id = id + 1;
VACUUM t_test;
```

尺寸还是没有改变。让我们看看表里有什么：

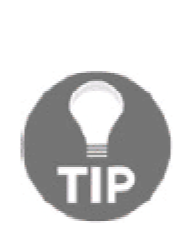
```
test=# SELECT ctid, * FROM t_test ORDER BY ctid DESC;
    ctid     |   id
-----------+--------
 (1327,46) | 112
 (1327,45) | 111
 (1327,44) | 110
 ...
 (884,20)  | 99798
 (884,19)  | 99797
 ...
```

ctid 是行在磁盘上的物理位置。通过使用 ORDER BY ctid DESC，用户将按照物理顺序从后向前读到该表。为什么要关心这一点？原因是在该表的末尾有一些非常小的值和非常大的值。如果它们被删除会发生什么？

```
test=# DELETE FROM t_test WHERE id > 99000 OR id < 1000;
DELETE 1999
test=# VACUUM t_test;
VACUUM
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
```

```
 pg_size_pretty
----------------
 3504 kB
(1 row)
```

尽管只有 2%的数据被删除，该表的尺寸却下降了三分之二。原因在于，如果 VACUUM 只在该表中的特定位置之后寻找死亡行，它可以把空间还给文件系统。这是唯一一种可以看到表尺寸下降的情况。当然，普通用户无法控制数据在磁盘上的物理位置。因此，存储消耗将很可能保持相同，除非所有的行都被删除。



为什么有这么多小值和大值在该表的末尾？在该表最初被 100000 个行填充之后，最后一个块并没有被完全填满，因此第一次的 UPDATE 将用更改填满最后一块。这自然把该表的末尾搅乱了一点。在这个精心布置的例子中，这就是在表末尾出现如此怪异布局的原因。

在真实世界的应用中，这种现象的影响没有得到足够的重视。没有真正理解存储是不可能进行性能调优的。

2.7.3　利用 snapshot too old

VACUUM 做得很好并且它将根据需要回收空闲空间。但是何时 VACUUM 才能真正清理掉行并且把它们转变成空闲空间？规则是这样：如果一行再也不能被任何人看见，它就能被回收。实际上这意味着最老的事务都看不见的部分就可以被认为是真正死亡了。

这还意味着真正的长事务可以把清理推迟相当长的时间，由此带来的后果就是表膨胀。表将会超比例增长并且性能将会趋向于衰退。

幸运地是，PostgreSQL 9.6 有一个很好的特性允许管理员聪明地限制一个事务的时间。Oracle 管理员会非常熟悉 snapshot too old 错误。从 PostgreSQL 9.6 开始，也有了这种错误消息。但是它更像是一种特性，而不是由于错误配置而导致的意外的副作用（Oracle 中就是这样）。

为了限制快照的生存时间，可以使用 postgresql.conf 中的一个设置：

```
old_snapshot_threshold = -1
          # 1min-60d; -1 disables; 0 is immediate
```

如果设置了这个变量，事务将会在一定量的时间之后失败。注意这个设置是在实例级别，并且不能在会话中设置。通过限制一个事务的尺寸，超长事务带来的风险将会急剧下降。

2.8　总　　结

在本章中，读者学到了有关事务、锁定及其逻辑含义，以及 PostgreSQL 事务机制用于存储的一般架构，还有相关的管理。读者也看到了行是如何被锁定的以及存在哪些特性。

在第 3 章中，读者将学习数据库工作中最重要的主题——索引。读者会学到关于 PostgreSQL 查询优化器的内容以及几种索引类型和它们的行为。

第 3 章　使 用 索 引

在第 2 章中，读者学到了有关并发和锁定的内容。在本章中我们将直面索引。这一主题的重要性怎么强调也不过分，索引是（并且很可能将总是）每一个数据库工程师职业生涯中最重要的主题之一。

在经历了 17 年专业的全职 PostgreSQL 咨询和 24×7 技术支持生涯之后，笔者可以肯定一件事情：糟糕的索引是糟糕性能的主因。当然，调整内存参数等也很重要。但是，如果索引没有被正确使用，一切都将是徒劳。一个缺失索引是无可替代的。因此，笔者将一整章的篇幅单独留给索引，以便让读者尽可能多地了解索引。

在本章中，读者将学到下面这些主题：

- 什么时候 PostgreSQL 使用索引？
- 优化器如何工作？
- 有些什么类型的索引以及它们怎么工作？
- 使用你自己的索引策略。

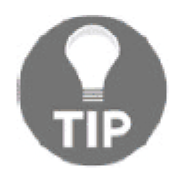
在结束本章后，读者将能理解在 PostgreSQL 中如何有效地使用索引。

3.1　理解简单查询和代价模型

在本节中，我们将开始使用索引。为了便于展示，我们需要一些测试数据。下面的代码片段展示如何轻易地创建这些数据：

```
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name) SELECT 'hans'
    FROM generate series(1, 2000000);
INSERT 0 2000000
test=# INSERT INTO t test (name) SELECT 'paul'
    FROM generate series(1, 2000000);
INSERT 0 2000000
```

第一行创建了一个简单的表，其中用到了两个列：一个自增列，它保存增长着的数字，另一个列则用静态值填充。



generate_series 函数将生成从 1 到 2 百万的数字。因此在这个例子中，会为 hans 和 paul 都创建 2 百万个静态值。

总计有 4 百万行被增加到表中：

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
 name |  count
------+---------
 hans | 2000000
 paul | 2000000
(2 rows)
```

这 4 百万行有一些很好的性质：ID 是升序的并且只有两种不同的名字。

现在让我们运行一个简单的查询：

```
test=# \timing
Timing is on.
test=# SELECT * FROM t_test WHERE id = 432332;
   id   | name
--------+------
 432332 | hans
(1 row)

Time: 119.318 ms
```

在这种情况下，\timing 命令将告诉 psql 显示一个查询的运行时间。注意这不是在服务器上真正的执行时间，而是 psql 测量的时间。在查询非常短的情况下，网络延迟可能会是总时间中占比大的组成部分，因此必须被考虑在内。

3.1.1　使用 EXPLAIN

在这个例子中，读取 4 百万行用了超过 100 毫秒的时间。从性能的角度来看，这完全是一场灾难。为了找出什么地方不对，PostgreSQL 提供了 EXPLAIN 命令：

```
test=# \h EXPLAIN
Command:     EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

   ANALYZE [ boolean ]
   VERBOSE [ boolean ]
   COSTS [ boolean ]
```

```
   BUFFERS [ boolean ]
   TIMING [ boolean ]
   FORMAT { TEXT | XML | JSON | YAML }
```

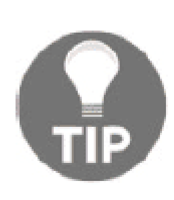
当感觉一个查询执行得不好时，EXPLAIN 将帮助我们揭示真正的性能问题。它是这样工作的：

```
test=# EXPLAIN SELECT * FROM t test WHERE id = 432332;
                               QUERY PLAN
---------------------------------------------------------------
 Gather (cost=1000.00..43463.92 rows=1 width=9)
   Workers Planned: 2
   -> Parallel Seq Scan on t_test
       (cost=0.00..42463.82 rows=1 width=9)
        Filter: (id = 432332)
(4 rows)
```

读者在这个列表中看到的就是所谓的执行计划。在 PostgreSQL 中，一个 SQL 语句将被分成 4 个阶段执行。下列组件会参与其中：

- 解析器将检查语法错误以及明显的问题。
- 重写系统负责规则（视图等）。
- 优化器将解决如何以最有效的方法执行一个查询并且制订出一个计划。
- 优化器提供的计划将被执行器用来最终创建结果。

EXPLAIN 的目的是看看计划器给出什么样的东西来高效地运行查询。在笔者的例子中，PostgreSQL 将使用一个并行顺序扫描，这意味着两个工作者将合作来处理过滤条件。得到的局部结果接下来通过一个称为收集节点的东西联合起来，收集节点在 PostgreSQL 9.6 中作为并行查询架构的一部分引入。如果读者更加细致地观察这个计划，将会看到 PostgreSQL 在该计划的每个阶段预期得到的行数（在这个例子中，rows = 1，即将返回 1 行）。



在 PostgreSQL 9.6 中，并行工作者的数量将由表的尺寸决定。一个操作越大，PostgreSQL 就将发动更多的并行工作者。对于一个非常小的表，并行机制不会被使用，因为它会带来太多开销。

并行并非必不可少，通过将下面的变量设置为 0，总是可以将并行工作者的数量降低成 PostgreSQL 9.6 之前的行为：

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
```

注意这种改变没有副作用，因为它只在用户的会话中有效。当然用户也可以决定在 postgresql.conf 中更改该变量，但笔者不建议这样做，因为那样会损失很多由并行查询带来的性能。

3.1.2　深究 PostgreSQL 代价模型

如果只有一个 CPU 被使用，执行计划看起来将会像这样：

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on t_test (cost=0.00..71622.00 rows=1 width=9)
   Filter: (id = 432332)
(2 rows)
```

PostgreSQL 将顺序读取（顺序扫描）整个表并且应用过滤条件。它预计该操作会花费 71622 惩罚点。这是什么意思呢？惩罚点（或者说代价）通常是一种抽象概念。它们被用来比较执行查询的不同方式。如果一个查询可以被执行器以很多不同的方式执行，PostgreSQL 将选取承诺最低代价的执行计划。现在的问题是：PostgreSQL 是怎样得到 71622 点的？

原理如下：

```
test=# SELECT pg relation size('t test') / 8192.0;
      ?column?
--------------------
 21622.000000000000
(1 row)
```

pg_relation_size 函数将会返回该表以字节为单位的大小。在给定的例子中，可以看到这个关系由 21622 个块（每个大约 8000 字节）构成。根据代价模型，PostgreSQL 将会为它必须顺序读取的每一个块加上代价 1。

影响这个过程的配置参数是：

```
test=# SHOW seq page cost;
 seq page cost
---------------
 1
(1 row)
```

不过，从磁盘读取一大堆块并非我们要做的所有事情。还需要通过 CPU 应用过滤条

件并且发送那些行。这里有两个参数负责相应的代价：

```
test=# SHOW cpu_tuple_cost;
 cpu_tuple_cost
----------------
 0.01
(1 row)
test=# SHOW cpu operator cost;
 cpu operator cost
-------------------
 0.0025
(1 row)
```

这会导致下面的计算：

```
test=# SELECT 21622*1 + 4000000*0.01 + 4000000*0.0025;
  ?column?
------------
 71622.0000
(1 row)
```

如你所见，这正好是我们在计划中看到的数字。代价由一个 CPU 的部分和一个 I/O 的部分组成，它们都将被转变成为一个单一的数字。这里的重点是代价与实际执行没有关系，因此不可能把代价解读成毫秒。计划器给出的数字仅仅只是一个估计值。

当然，还有比这个简短例子中提到的参数更多的参数。PostgreSQL 还有一些用于索引相关操作的特殊参数。

- random_page_cost = 4：如果 PostgreSQL 使用一个索引，通常会涉及很多随机 I/O。在传统的旋转型磁盘上，随机读比顺序读重要得多，因此 PostgreSQL 也将相应地解释它们。注意在 SSD 上，随机读和顺序读之间的差异已经不再存在，因此可以在 postgresql.conf 文件中设置 random_page_cost = 1。
- cpu_index_tuple_cost = 0.005：如果使用了索引，PostgreSQL 还将考虑一些索引的 CPU 代价。

如果用户使用并行查询，还有更多代价参数。

- parallel_tuple_cost = 0.1：这个参数定义了从一个并行工作者进程向另一个进程传输一个元组的代价。它基本上用于解释在并行架构内部移动元组的开销。
- parallel_setup_cost = 1000.0：这个参数调整发动一个工作者进程的代价。当然，启动进程来并行运行查询不是免费的，因此这个参数试图建模那些与进程管理相关的代价。

- min_parallel_relation_size = 8 MB：这个参数定义了考虑使用并行查询的表的最小尺寸。一个表长得越大，PostgreSQL 就将使用更多的 CPU。表的尺寸必须成为之前 3 倍才会多出一个工作者进程。

3.1.3　部署简单的索引

发动更多工作进程来扫描非常大的表有时候并不能解决问题。读取整个表只为寻找一行通常也不是什么好主意。因此，我们需要创建索引：

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
test=# SELECT * FROM t_test WHERE id = 43242;
   id  | name
-------+------
 43242 | hans
(1 row)
Time: 0.259 ms
```

PostgreSQL 使用 Lehman-Yao 的高并发性 B-树作为标准索引。连同一些 PostgreSQL 的专门优化一起，这些树为最终用户提供了优秀的性能。最重要的一点是，Lehman-Yao 让用户能在同一时间在同一个索引上运行很多操作（读和写），这有助于极大地提升吞吐量。

但是，索引也不是白用的：

```
test=# \di+
                       List of relations
 Schema |  Name  |  Type | Owner | Table  | Size  | Description
--------+--------+-------+-------+--------+-------+------------
 public | idx_id | index | hs    | t_test | 86 MB |
(1 row)
```

如你所见，我们的索引包含的 4 百万行，它将用掉 86MB 的磁盘空间。除此之外，对于该表的写入将会变慢，因为索引必须一直被保持同步。

3.1.4　使用排序输出

B-树索引并非只对查找行有用。它们也可用来给下一阶段处理提供排序好的数据：

```
test=# EXPLAIN SELECT * FROM t_test ORDER BY id DESC LIMIT 10;
                            QUERY PLAN
```

```
--------------------------------------------------------------
 Limit (cost=0.43..0.74 rows=10 width=9)
   -> Index Scan Backward using idx_id on t_test
         (cost=0.43..125505.43 rows=4000000 width=9)
(2 rows)
```

在这种情况中，索引已经按正确的顺序返回了数据，因此不需要再对整个数据集排序。读取索引的最后 10 行就足够回答这个查询了。实际上，这意味着可以在零点几毫秒内找到一个表中的前 N 行。

不过，ORDER BY 不是唯一要求排序输出的操作。min 和 max 函数也都与排序输出有关，因此索引也可以被用来加速这两个操作。这里有一个例子：

```
test=# explain SELECT min(id), max(id) FROM t test;
                           QUERY PLAN
---------------------------------------------------------------
 Result (cost=0.93..0.94 rows=1 width=8)
  InitPlan 1 (returns $0)
    -> Limit (cost=0.43..0.46 rows=1 width=4)
     -> Index Only Scan using idx_id on t_test
          (cost=0.43..135505.43 rows=4000000 width=4)
           Index Cond: (id IS NOT NULL)
  InitPlan 2 (returns $1)
    -> Limit (cost=0.43..0.46 rows=1 width=4)
     -> Index Only Scan Backward using idx id on t test t test 1
          (cost=0.43..135505.43 rows=4000000 width=4)
           Index Cond: (id IS NOT NULL)
(9 rows)
```

在 PostgreSQL 中，一个索引（或者更准确点说是一棵 B-树）可以以正常的顺序或者反向顺序被读取。现在的情况是：一棵 B-树可以被看成是一个已排序的列表。因此自然是最小值在开头而最大值在结尾。因此，min 和 max 非常适合用索引来加速。

在 SQL 中，很多操作依赖于排序输入。因此，理解那些操作是必要的，因为它们会对索引有很多严重的影响。

3.1.5 一次使用多个索引

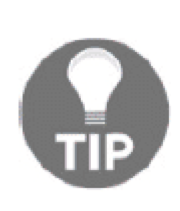
到目前为止，读者已经看到了一次使用一个索引的例子。不过，在很多真实世界的情况中，这还远远不够。有些情况会要求在数据库中使用更多的逻辑。

PostgreSQL 允许在单个查询中使用多个索引。当然，如果同时有很多列被查询这才有意义。但并非总是如此，也可能会发生同一个索引被多次使用来处理同一列的情况。

这里有一个例子：

```
test=# explain SELECT * FROM t_test WHERE id = 30 OR id = 50;
                           QUERY PLAN
-----------------------------------------------------------
 Bitmap Heap Scan on t test (cost=8.88..16.85 rows=2 width=9)
   Recheck Cond: ((id = 30) OR (id = 50))
   -> BitmapOr (cost=8.88..8.88 rows=2 width=0)
     -> Bitmap Index Scan on idx idv (cost=0.00..4.44 rows=1 width=0)
           Index Cond: (id = 30)
     -> Bitmap Index Scan on idx_id (cost=0.00..4.44 rows=1 width=0)
           Index Cond: (id = 50)
(7 rows)
```

这里的重点是有两个地方需要 id 列。首先该查询会找 30，然后又会找 50。如你所见，PostgreSQL 将会进行所谓的位图扫描。



位图扫描和位图索引（有 Oracle 背景的人可能会知道）并不相同。它们是两种完全不同的概念；并且没有任何共同点。位图索引是 Oracle 中的一种索引类型，而位图扫描基本上是一种扫描方法。

在位图扫描背后的思想是，PostgreSQL 将首先扫描第一个索引，收集含有数据的块的一个列表。然后第二个索引被扫描并且同样得到一个块列表。有多少个索引就会这样做多少次。在 OR 的情况下，这些列表会被统一，留给我们一个由含有数据的块构成的大列表。最后才会使用这个列表来扫描表以检索出那些块。现在的问题是，PostgreSQL 已经检索出的数据比需要的更多。在我们的情况中，查询将查找两行，但是位图扫描可能返回的是几个块。因此，执行器将进行所谓的复查来过滤掉那些不满足条件的行。

位图扫描也可以用于 AND 条件或者 AND 与 OR 的混合条件。不过，如果 PostgreSQL 看到一个 AND 条件，它不一定会强制用位图扫描。让我们假设有一个查询查找每一个生活在奥地利的人以及一个有特定 ID 的人。这里使用两个索引实际上没有意义，因为在搜索该 ID 后实际上不会有多少数据剩下来。扫描两个索引会是更加昂贵的方法，因为有 8 百万人（包括我自己）生活在奥地利，从性能的观点来看，把这么多行读进来就为了查找一个人实在是没有意义。好消息是 PostgreSQL 的优化器会通过比较不同选项和可能索引的代价来为用户做出所有这些决定，因此无须为此感到担忧。

- 有效地使用位图扫描

这时候很自然地会出现一个问题：什么时候一个位图扫描最有用并且什么时候优化器才会选择它？在笔者看来，实际上有两种用例：

- 避免反复地使用同一个块。
- 组合相对不太好的条件。

第一种情况相当普遍。假定用户正在查找讲一种特定语言的每一个人。为了举例方便，假定所有人中有 10%讲所要求的语言。扫描索引意味着表中的一个块必须被一次又一次地扫描，因为很多说这种语言的人可能被存储在同一块中。通过应用一个位图扫描，可以确保一个特定的块只被使用一次，这当然能够导致更好的性能。

第二种常见的用例是将相对较弱的条件用在一起。让我们假设要查找年龄介于 20～30 岁且拥有一件黄色衬衫的每一个人。现在，可能所有人中有 15%介于 20～30 岁，并且有 15%拥有一件黄色衬衫。顺序扫描一个表是非常昂贵的，因此 PostgreSQL 可能会决定选择两个索引，因为最终的结果可能只由 1%的数据构成。扫描这两个索引可能比读取所有数据要划算。

3.1.6　以一种聪明的方式使用索引

到目前为止，应用一个索引就好像圣杯，它总是能神奇地提升性能。不过，并非总是如此。索引在某些情况下也会相当没有意义。

在更加深入探究之前，这里有一个用于这个例子的数据结构。记住只有两个不同的名字和唯一的 ID：

```
test=# \d t_test
                Table "public.t test"
 Column |   Type   |  Modifiers
--------+----------+----------------------------------
 id     | integer  | not null default nextval('t test id seq'::regclass)
 name   | text     |
Indexes:
    "idx_id" btree (id)
```

此时，已经有一个索引被定义，它覆盖了 id 列。在下一步中将查询 name 列。在查询前，将先在 name 上创建一个索引：

```
test=# CREATE INDEX idx_name ON t_test (name);
CREATE INDEX
```

现在是时候看看该索引是否被正确使用了：

```
test=# EXPLAIN SELECT * FROM t test WHERE name = 'hans2';
                      QUERY PLAN
-------------------------------------------------------------
```

```
 Index Scan using idx_name on t_test (cost=0.43..4.45 rows=1 width=9)
   Index Cond: (name = 'hans2'::text)
(2 rows)
```

如我们所愿，PostgreSQL 将决定使用该索引。这也是大部分用户期待的。但注意笔者的查询说的是 hans2。记住：hans2 在该表中不存在，并且查询计划完美地反映了这一点。rows=1 表示计划器只期待该查询返回数据的一个非常小的子集。



在表中一个行也没有，但是 PostgreSQL 从不会估计出零行，因为那会让后续的估计变得困难许多。

如果我们查找更多数据，来看看会发生什么：

```
test=# EXPLAIN SELECT * FROM t_test WHERE name='hans' OR name='paul';
                          QUERY PLAN
--------------------------------------------------------------
 Seq Scan on t_test (cost=0.00..81622.00 rows=3000011 width=9)
   Filter: ((name = 'hans'::text) OR (name = 'paul'::text))
(2 rows)
```

在这种情况下，PostgreSQL 将使用一个直截了当的顺序扫描。为什么会这样？为什么系统忽略了所有的索引？原因很简单：hans 和 paul 组成了整个数据集，因为其中没有其他值。因此，PostgreSQL 认为整个表无论怎样都必须被读取。如果只需要读取表就足够，那么就没有理由去读取所有索引以及整个表。

换句话说，PostgreSQL 并不会只因为有一个索引而使用它。PostgreSQL 会在索引有意义时使用。如果行数较小，PostgreSQL 将再次考虑位图扫描和普通索引扫描：

```
test=# EXPLAIN SELECT * FROM t_test WHERE name = 'hans2' OR name =
'paul2';
                            QUERY PLAN
----------------------------------------------------------
 Bitmap Heap Scan on t_test (cost=8.88..12.89 rows=1 width=9)
  Recheck Cond: ((name = 'hans2'::text) OR (name = 'paul2'::text))
  -> BitmapOr (cost=8.88..8.88 rows=1 width=0)
     -> Bitmap Index Scan on idx_name (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (name = 'hans2'::text)
     -> Bitmap Index Scan on idx name (cost=0.00..4.44 rows=1 width=0)
          Index Cond: (name = 'paul2'::text)
```

这里要学习的最重要的一点是执行计划取决于输入值。它们不是静态的并且依赖于

表中的数据。这是一个非常重要的观察，我们要一直把它记在脑子里。在真实世界的例子中，计划的改变可能常常导致不可预知的运行时间。

3.2　使用聚簇表改善速度

在本节中，读者将认识到关联的力量以及聚簇表的力量。其关键思想是什么？考虑用户想要读取一整个范围的数据的情况。这个范围可能是一定的时间范围、一些块、ID 等。这种查询的运行时间会根据数据量以及数据在磁盘上的物理分布而变化。因此，即使用户运行返回同样数量行的查询，两个系统可能不会在相同的时间跨度内提供回答，因为物理磁盘布局可能会不同。

这里有一个例子：

```
test=# EXPLAIN (analyze true, buffers true, timing true)
  SELECT *
  FROM t test
  WHERE id < 10000;
                              QUERY PLAN
-----------------------------------------------------------------
 Index  Scan  using  idx_id  on  t_test  (cost=0.43..370.87  rows=10768
width=9)
(actual time=0.011..2.897 rows=9999 loops=1)
   Index Cond: (id < 10000)
   Buffers: shared hit=85
 Planning time: 0.078 ms
 Execution time: 4.081 mundefined
(5 rows)
```

读者可能还记得，数据是以一种有组织的并且顺序的方式被装载到表中。数据被一个 ID 接着一个 ID 添加，因此可以预期这些数据将以连续的顺序放在磁盘上。如果数据使用某个自增列被装载到一个空表中，情况就是如此。

我们已经见过 EXPLAIN 的作用，这个例子中利用了 EXPLAIN (analyze true, buffers true, timing true)。analyze 除了展示查询计划之外，还会执行该查询并且展示执行时的情况。对比较规划器的估计值和实际值来说，EXPLAIN analyze 非常好。要判断规划器是正确的还是差得很远，这是一种最好的方法。buffers true 参数将告诉我们该查询会用到多少 8000 字节的块，在这个例子中，总共用了 85 个块。shared hit 表示来自于 PostgreSQL 的 I/O 缓冲（共享缓存）的数据，PostgreSQL 一共花了 4 毫秒来检索这些数据。

如果表中的数据有点随机会怎样呢？会发生变化吗？要创建一个含有同样的数据但

顺序随机的表，可以简单地使用 ORDER BY random()。它将确保数据在磁盘上确实是混乱的：

```
test=# CREATE TABLE t_random AS SELECT * FROM t_test ORDER BY random();
SELECT 4000000
```

为了保证比较公平，在同一个列上建立索引：

```
test=# CREATE INDEX idx_random ON t_random (id);
CREATE INDEX
```

要正确工作，PostgreSQL 会需要优化器统计信息。那些统计信息将会告诉 PostgreSQL 有多少数据、值是怎样分布以及数据在磁盘上是否相关。为了更快地得到统计信息，笔者在这里增加了一次 VACUUM 调用。请注意，关于 VACUUM 稍后将在本书中详细讨论：

```
test=# VACUUM ANALYZE t_random;
VACUUM
```

现在让我们运行同样的查询：

```
test=# EXPLAIN (analyze true, buffers true, timing true) SELECT * FROM
t_random WHERE id < 10000;
                            QUERY PLAN
----------------------------------------------------------------
 Bitmap Heap Scan on t random
         (cost=203.27..18431.86 rows=10689 width=9)
         (actual time=5.087..13.822 rows=9999 loops=1)
   Recheck Cond: (id < 10000)
   Heap Blocks: exact=8027
   Buffers: shared hit=8057
   -> Bitmap Index Scan on idx_random
          (cost=0.00..200.60 rows=10689 width=0)
          (actual time=3.558..3.558 rows=9999 loops=1)
         Index Cond: (id < 10000)
         Buffers: shared hit=30
 Planning time: 0.075 ms
 Execution time: 14.411 ms
(9 rows)
```

这里可以观察到很多事情。首先，需要的数据块是令人震惊的 8057 块，而且运行时间也飞涨到 14 毫秒。唯一在某种程度上拯救了性能的事情是，数据还是来自于内存而非磁盘。想想如果不得不访问磁盘 8057 次来回答这个查询意味着什么，那将是一场彻头彻尾的灾难，因为磁盘等待势必会让速度剧烈地慢下来。

不过，还可以看到更多，甚至可以看到计划已经发生了改变。PostgreSQL 现在使用一个位图扫描取代了普通索引扫描。这被用来减少该查询中所需的块数以防止更差的行为发生。

规划器是怎么知道数据被如何存储在磁盘上？pg_stats 是一个包含了所有有关列内容统计信息的系统视图。下面的查询显示了有关的内容：

```
test=# SELECT tablename, attname, correlation FROM pg_stats WHERE
tablename IN ('t_test', 't_random') ORDER BY 1, 2;
 tablename | attname | correlation
-----------+---------+-------------
 t_random  | id      | -0.0114944
 t_random  | name    | 0.493675
 t_test    | id      | 1
 t_test    | name    | 1
(4 rows)
```

可以看到 PostgreSQL 关心每一个列。该视图的内容由一种叫作 ANALYZE 的东西创建，它对性能至关重要：

```
test=# \h ANALYZE
Command:     ANALYZE
Description: collect statistics about a database
Syntax:
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

通常，会有一个所谓的 autovacuum 守护进程在后台自动地执行 ANALYZE，autovacuum 会在本书的后续部分中介绍。

回到查询。如你所见，两个表都有两列（id 和 name）。对于 t_test.id，关联度是 1，这表示下一个值有些依赖于前一个值。在笔者的例子中，数字是升序的。同样的情况也适用于 t_test.name。首先，有一些项含有 hans；其次还有一些值含有 paul。所有相同的名称因此被存在一起。

在 t_random 中，情况大相径庭，一个负的相关度表示数据是混乱的。还可以看到 name 列的关联度大约是 0.5，实际上，它表示在该表中相同的名字通常不是挨在一起，它还意味着以物理顺序读取该表时，读到的名字总是在切换。

为什么这会导致该查询命中这么多块？答案相对比较简单。如果我们需要的数据并没有被紧密地包装在一起而是均匀地散布在表中，就需要更多的块来提取出等量的信息，进而导致更糟糕的性能。

3.2.1　聚簇表

在 PostgreSQL 中，有一个叫作 CLUSTER 的命令，它让我们能够以一种想要的顺序重写一个表。可以指定一个索引并且按该索引的顺序来存储数据：

```
test=# \h CLUSTER
Command:     CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

CLUSTER 命令已经存在了很多年并且效果良好。但是，盲目地在一个生产系统中运行它之前，有一些事情需要考虑：

- CLUSTER 命令在运行时将锁住表。在 CLUSTER 运行期间，不能插入或者修改数据。这对于生产系统可能是无法接受的。
- 数据只能被按照一个索引进行组织。用户不能同时通过邮编、姓名、ID、生日等排序同一个表。这意味着如果存在一个大部分时间都会被使用的搜索条件，CLUSTER 就有意义。
- 记住本书中给出的例子更像是一种最坏的情况。实际上，一个聚簇表和一个未聚簇表之间的性能差异将取决于负载、接收到的数据量、缓存命中率等很多因素。
- 如果一个表在正常操作时被更改，它的聚簇状态将无法维持。随着时间的流逝，关联度可能会恶化。

这里是一个如何运行 CLUSTER 命令的例子：

```
test=# CLUSTER t_random USING idx_random;
CLUSTER
```

聚簇所需的时间会根据表的尺寸而变化。

3.2.2　使用只用索引的扫描

到目前为止，读者已经看到了何时会使用索引以及何时不会使用索引。除此之外，还讨论了位图扫描。

不过，索引还不止这些内容。下面的两个例子只有很小的区别，但是它们的性能差异可能会相当大。这是第一个查询：

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 34234;
                           QUERY PLAN
----------------------------------------------------------------
 Index Scan using idx_id on t_test
    (cost=0.43..8.45 rows=1 width=9)
    Index Cond: (id = 34234)
```

这里看不出有什么不寻常的。PostgreSQL 使用一个索引来寻找一个行。那么如果只选择一个列会发生什么呢？

```
test=# EXPLAIN SELECT id FROM t_test WHERE id = 34234;
                         QUERY PLAN
----------------------------------------------------------------
 Index Only Scan using idx_id on t_test
    (cost=0.43..8.45 rows=1 width=4)
    Index Cond: (id = 34234)
(2 rows)
```

如你所见，计划已经从一个索引扫描变成了一个所谓的只用索引的扫描。在我们的例子中，id 列已经被建立索引，因此它的内容很自然地存在于该索引中。如果所有的数据已经可以从索引中取出，那么在大部分情况下就没有必要去表中取。（几乎）只有额外的列被查询时，才需要去表中取数据，而这个例子并不属于这种情况。因此，只用索引的扫描将会带来比普通索引扫描明显更好的性能。

实际上，甚至可以在一个索引中包括额外列来享受这一特性带来的好处。在 MS SQL 中，增加额外列被称为**覆盖索引**。PostgreSQL 也能实现类似的行为。

3.3　理解另外的 B-树特性

在 PostgreSQL 中，索引是一个很大的领域并且覆盖了数据库工作的很多方面。正如笔者在本书中已经介绍的，索引是性能的关键，没有正确的索引就不会有好的性能。因此，那些与索引相关的特性值得更详细地检阅。

3.3.1　组合索引

在作为一个专业 PostgreSQL 支持提供商的日子里，笔者常常被问到组合索引和个体索引之间的区别。在本节中，笔者将尝试弄清楚这个问题。

一般的规则是这样：如果单个索引就能回答用户的问题，它通常就是最好的选择。不过，不可能在人们过滤的域的所有可能组合上建立索引。我们所能做的就是使用组合

索引的性质来达到尽可能多的收益。

假定有一个含有 3 列（postal_code、last_name 和 first_name）的表。一个电话本会利用包含这 3 列的组合索引。读者将会看到其中的数据被根据位置排序，在同一个位置，数据将被按照姓和名来排序。

表 3-1 展示了哪些操作可能用到给定的 3 列索引。

表 3-1

查　询	是否可能	备　注
postal_code = 2700 AND last_name = 'Schönig' AND first_name = 'Hans'	可能	这是这个索引最理想的用例
postal_code = 2700 AND last_name = 'Schönig'	可能	没有限制
last_name = 'Schönig AND postal_code = 2700	可能	PostgreSQL 将简单地交换条件
postal_code = 2700	可能	这就像在 postal_code 上的一个索引，这个组合索引只是需要磁盘上的更多空间
first_name = 'Hans'	可能，但是是一种不同的用例	PostgreSQL 不再能使用该索引的排序性质。不过，在一些极端情况（通常是非常宽的表，包括无数列）中，如果扫描整个索引和读取这个非常宽的表代价一样低，PostgreSQL 会选择扫描整个索引

如果列被单独索引，用户最后看到的将很有可能是位图扫描。当然，一个单一的定制索引会更好。

3.3.2　增加函数索引

到目前为止，读者已经见到了如何原封不动地索引列的内容。但是，这并非总是用户真正想要的。因此，PostgreSQL 允许创建函数索引。基本思想非常简单，索引并不直接包括值，而是存储一个函数的输出。

下面的例子展示了如何索引 id 列的余弦值：

```
test=# CREATE INDEX idx_cos ON t_random (cos(id));
CREATE INDEX
test=# ANALYZE;
ANALYZE
```

我们要做的仅仅是把函数放在索引创建命令的列列表中。当然，并不是对所有种类的函数都能这样做。只有输出不变的函数才能被这样使用：

```
test=# SELECT age('2010-01-01 10:00:00'::timestamptz);
           age
-------------------------
 6 years 9 mons 14:00:00
(1 row)
```

age 之类的函数实在不适合用作索引，因为它们的输出不是常数。随着时间的流逝，age 的输出也会改变。PostgreSQL 会明确地禁止对给定的相同输入可能改变结果的函数。从这个方面来看，余弦函数是可以的，因为一个值的余弦值在 1000 年之内都是相同的。

要测试这个索引，笔者写了一个简单的查询来展示：

```
test=# EXPLAIN SELECT * FROM t_random WHERE cos(id) = 10;
                           QUERY PLAN
--------------------------------------------------------------
 Index Scan using idx_cos on t_random (cost=0.43..8.45 rows=1 width=9)
   Index Cond: (cos((id)::double precision) = '10'::double precision)
(2 rows)
```

如我们所愿，函数索引会像任何其他索引一样被使用。

3.3.3 减少空间消耗

索引是好东西，它的主要目的是尽可能提高速度。但是和所有的好东西一样，索引也是有代价的，即空间消耗。为了发挥其魔力，索引必须以一种有组织的方式存储值。如果用户的表含有一千万个整数值，属于该表的索引在逻辑上将会包含这一千万个整数值。

B-树还将包含指向表中每一个行的指针，因此索引并非是免费的。为了看到一个索引需要多少空间，可以使用\di+命令问问 psql：

```
test=# \di+
                           List of relations
 Schema |    Name     | Type  | Owner |   Table   | Size
--------+-------------+-------+-------+-----------+-------
 public | idx_cos     | index | hs    | t_random  | 86 MB
 public | idx_id      | index | hs    | t_test    | 86 MB
 public | idx_name    | index | hs    | t_test    | 86 MB
 public | idx_random  | index | hs    | t_random  | 86 MB
(4 rows)
```

在笔者的数据库中，存储那些索引消耗掉了令人吃惊的 344MB。现在，来与底层表烧掉的存储量做个比较：

```
test=# \d+
                          List of relations
 Schema |      Name      |   Type   | Owner |    Size
--------+----------------+----------+-------+------------
 public | t random       | table    | hs    | 169 MB
 public | t test         | table    | hs    | 169 MB
 public | t test id seq  | sequence | hs    | 8192 bytes
(3 rows)
```

两个表的尺寸加起来也只不过是 338MB。换句话说，我们的索引需要比实际数据更多的空间。在现实世界中，这种现象很普遍并且确实很容易出现。最近笔者拜访了一个在德国的 Cybertec 的客户，笔者看到了一个数据库，其大小的 64%都由从不使用（在数月间都没有使用哪怕一次）的索引构成。因此，过度索引也会和索引不足一样成为一个问题。回忆一下，那些索引不只是消耗空间，每一个 INSERT 或者 UPDATE 也必须维护那些索引中的值。在像我们的例子的极端情况下，这会极大地降低写吞吐量。

如果在表中只有少数不同的值，部分索引是一种方案：

```
test=# DROP INDEX idx name;
DROP INDEX
test=# CREATE INDEX idx name ON t test (name) WHERE name NOT IN ('hans',
'paul');
CREATE INDEX
```

在这种情况下，大部分数据被排除在该索引之外，这样可以享受到一个小的、高效的索引：

```
test=# \di+ idx_name
                        List of relations
 Schema |   Name   | Type  | Owner | Table  |    Size
--------+----------+-------+-------+--------+-----------
 public | idx name | index | hs    | t test | 8192 bytes
(1 row)
```

注意只有排除那些非常频繁的、占据了表中很大一部分（至少 25%左右）的值才有意义。部分索引的理想候选是性别（我们假设大部分人是男性或者女性）、国籍（假设大部分人都有相同的国籍）等。当然，应用这类花招需要对数据有深入的认识，但它一定会有效果。

3.3.4　在建立索引时添加数据

创建一个索引很容易。但是，在索引正在被构建时不能修改其基表。CREATE INDEX 命令将会用一个 SHARE 锁锁住表来确保不会发生任何更改。虽然这对于小表不成问题，但它将会在生产系统的大型表上导致问题。索引 TB 大小的数据将会花一些时间并且因此会阻塞该表过长时间，这就是一个问题。

该问题的解决方案是 CREATE INDEX CONCURRENTLY 命令。这个命令构建索引将会花掉更长的时间（通常至少是普通创建的两倍），但用户可以在创建索引期间正常地使用该表。

下面是其执行方式：

```
test=# CREATE INDEX CONCURRENTLY idx_name2 ON t_test (name);
CREATE INDEX
```

注意，如果使用 CREATE INDEX CONCURRENTLY 命令，PostgreSQL 不保证它会成功。如果在系统上运行的操作出于某种原因与索引创建发生冲突，创建结束后会得到一个被标记为无效的索引。

3.4　引入操作符类

到目前为止，我们的目标都是搞清楚应该索引什么以及盲目地在这个列或者列组上应用索引。其实，我们已经悄然接受了一个让我们能这样做的假设。迄今为止，我们所做的事情都基于这样一个假设，即数据被排序的顺序比较固定。事实上，这种假设可能不成立。当然，数字将总是有相同的顺序，但是其他种类的数据将很有可能没有一种预定义的、固定的排序顺序。

为了证明这一观点，笔者编制了一个现实世界的例子。看看下面两个记录：

```
1118 09 08 78
2345 01 05 77
```

现在的问题是，这两个行能被正确地排序吗？可能可以，因为一个出现在另一个之前。但这是不对的，因为这两个行确实有一些隐藏的语义。读者在这里看到的是两个奥地利社会保险号码。09 08 78 实际意味着 August 9, 1978，而 01 05 77 实际表示 May 1, 1977。前 4 个数字由一个校验码和某种自增的三位数构成。因此实际上 1977 位于 1978 之前，我们可能要考虑交换这两行来达到想要的排序顺序。

问题是 PostgreSQL 并不知道这两行实际意味着什么。如果一个列被标记为文本，PostgreSQL 将会应用标准规则来排序文本。如果该列被标记为数字，PostgreSQL 将应用标准规则来排序数字。任何情况下它都不会使用前述的那么古怪的方式来排序。如果读者觉得在处理那些数字时只需要考虑笔者之前抛出的事实，那就错了。一年有多少个月？12 个？远非如此。在奥地利的社会保险系统中，这些数字可能容纳 14 个月？为什么？回想一下，三位数仅仅是一个自增值。麻烦在这里：如果一个移民或者难民没有有效的文书并且不知道他/她的生日，将会给他/她分配一个虚构的在第 13 个月的生日。在 1990 年的巴尔干战争中，奥地利为超过 115000 位难民提供了庇护。显然这个三位数是不够的并且会增加第 14 个月。现在，哪一种标准数据类型能够处理这种从 1970 年年初（这种社会保险号码结构被引入的时间）开始就存在的 COBOL 遗留数据？答案是没有。

为了以一种理智的方式处理用于特殊目的的域，PostgreSQL 提供了操作符类：

```
test=# \h CREATE OPERATOR CLASS
Command:     CREATE OPERATOR CLASS
Description: define a new operator class
Syntax:
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data type
  USING index method [ FAMILY family name ] AS
  { OPERATOR strategy number operator name [ ( op type, op type ) ] [ FOR
SEARCH | FOR ORDER BY sort family name ]
  | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name (
argument_type [, ...] )
  | STORAGE storage_type
} [, ... ]
```

一个操作符类将会告诉一个索引应该怎么运转。让我们看一个标准的二叉树。它可以执行 5 种操作，如表 3-2 所示。

表 3-2

策　略	操 作 符	描　述
1	<	小于
2	<=	小于等于
3	=	等于
4	>=	大于等于
5	>	大于

标准的操作符类支持这本书中我们已经用过的标准数据类型和标准操作符。如果想

要处理社会保险号码，就需要提出自己的操作符来提供所需的逻辑。这些自定义操作符接着可以被用来构造一个操作符类，它其实就是一种传递给索引的策略，它可以设定索引应该怎样运作。

- 为 B-树设计一种操作符类

为了给出一个例子展示一个操作符类长什么样子，笔者已经设计了一些代码来处理社会保险号码。为了让代码更加简洁，笔者没有理会校验码之类的细节。

1. 创建新操作符

第一件事情，也是必须要做的一件事情，是设计想要的操作符。注意需要 5 种操作符。每一种策略一种操作符。索引的策略实际上像是一种插件，允许用户能植入自己的代码。

在开始之前，笔者已经编制了一些测试数据：

```
CREATE TABLE t_sva (sva text);

INSERT INTO t_sva VALUES ('1118090878');
INSERT INTO t_sva VALUES ('2345010477');
```

现在测试数据已经有了，是时候创建一个操作符了。为此，PostgreSQL 提供了 CREATE OPERATOR 命令：

```
test=# \h CREATE OPERATOR
Command:     CREATE OPERATOR
Description: define a new operator
Syntax:
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

基本上，其中的概念是这样的：一个操作符调用一个函数，函数会得到一个或者两个参数，一个是操作符的左参数而另一个是操作符的右参数。

如你所见，一个操作符无非就是一个函数调用。因此接下来就需要在那些隐藏在操作符下面的函数中实现所需要的逻辑。为了固定排序顺序，笔者已经写好了一个名为

normalize_si 的函数：

```
CREATE OR REPLACE FUNCTION normalize_si(text) RETURNS text AS $$
        BEGIN
        RETURN substring($1, 9, 2) ||
               substring($1, 7, 2) ||
               substring($1, 5, 2) ||
               substring($1, 1, 4);
        END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

调用该函数将会返回下面的结果：

```
test=# SELECT normalize_si('1118090878');
 normalize_si
--------------
 7808091118
(1 row)
```

如上所示，我们所做的也不过是交换了一些数位，现在就可以使用普通的字符串排序顺序了。在下一步中，这个函数就已经被用来直接比较社会保险号码了。第一个需要的函数是小于函数，它属于第一个策略：

```
CREATE OR REPLACE FUNCTION si lt(text, text) RETURNS boolean AS $$
        BEGIN
        RETURN normalize si($1) < normalize si($2);
        END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

这里有两件需要注意的事情：

- 函数不能用 SQL 写，只能用一种过程语言或者编译语言来编写函数。其原因在于 SQL 函数在某些情况下可能是内联的，这会让所有的努力前功尽弃。
- 应该坚持本章中所使用的命名习惯，这种习惯被社区广泛接受。小于函数应该被称为_lt，小于等于函数应该被称为_le 等。

有了这些知识后，就可以定义我们未来的操作符所需的函数：

```
-- lower equals
CREATE OR REPLACE FUNCTION si le(text, text) RETURNS boolean AS $$
        BEGIN
        RETURN normalize_si($1) <= normalize_si($2);
        END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

```
-- greater equal
CREATE OR REPLACE FUNCTION si ge(text, text) RETURNS boolean AS $$
   BEGIN
   RETURN normalize si($1) >= normalize si($2);
   END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;

-- greater
CREATE OR REPLACE FUNCTION si_gt(text, text) RETURNS boolean AS $$
   BEGIN
   RETURN normalize si($1) > normalize si($2);
   END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

目前，已经定义了 4 个函数。用于相等操作符的第 5 个函数不是必需的。我们可以简单地用已有的操作符，因为相等并不依赖于排序顺序。

现在所有的函数都就位了，是时候定义那些操作符了：

```
-- define operators
CREATE OPERATOR <# ( PROCEDURE=si_lt,
                     LEFTARG=text,
                     RIGHTARG=text);
```

操作符的设计其实非常简单。操作符需要一个名字（在笔者的情况中是<#)、一个要被调用的过程，还有左右参数的数据类型。当操作符被调用时，左参数将是 si_lt 的第一个参数，而右参数将是第二个参数。

余下的 3 个操作符遵循相同的原则：

```
CREATE OPERATOR <=# ( PROCEDURE=si le,
                      LEFTARG=text,
                      RIGHTARG=text);

CREATE OPERATOR >=# ( PROCEDURE=si_ge,
                      LEFTARG=text,
                      RIGHTARG=text);

CREATE OPERATOR ># ( PROCEDURE=si gt,
                     LEFTARG=text,
                     RIGHTARG=text);
```

所使用的索引类型还需要几个支持函数。在标准的 B-树中，只需要一个支持函数，

它被用来提高内部的速度：

```
CREATE OR REPLACE FUNCTION si_same(text, text) RETURNS int AS $$
        BEGIN
                IF      normalize_si($1) < normalize_si($2)
                THEN
                        RETURN -1;
                ELSIF  normalize_si($1) > normalize_si($2)
                THEN
                        RETURN +1;
                ELSE
                        RETURN 0;
                END IF;
        END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

如果第一个参数较小，si_same 函数将会返回-1，如果两者相等返回 0；如果第一个参数较大则返回 1。在内部，_same 函数的工作最重，因此应该确保优化其中的代码。

2. 创建操作符类

最后，所有的组件都已到位，可以创建索引所需的操作符类：

```
CREATE OPERATOR CLASS sva_special_ops
FOR TYPE text USING btree
AS
   OPERATOR     1      <# ,
   OPERATOR     2      <=# ,
   OPERATOR     3      = ,
   OPERATOR     4      >=# ,
   OPERATOR     5      ># ,

   FUNCTION     1      si_same(text, text);
```

CREATE OPERATOR CLASS 命令连接策略和操作符。“OPERATOR 1 <#”表示策略 1 将使用“<#”操作符。最后，_same 函数被连接到操作符类。

注意该操作符有一个名字并且被显式地定义为用于 B-树。

该操作符类已经可以在索引创建时使用：

```
CREATE INDEX idx_special ON t_sva (sva sva_special_ops);
```

创建的索引会以与之前略有不同的方式工作：sva sva_special_ops 表示使用 sva_special_

ops 操作符类来索引 sva 列。如果没有显式地使用 sva_special_ops，那么 PostgreSQL 将不会使用特殊的排序顺序而是使用默认的操作符类。

3. 测试自定义操作符类

在例子中，测试数据只有两行。因此，PostgreSQL 将不会使用索引，因为该表太小以至于使用索引的开销反而会更大。但是为了在不装载太多数据的情况下完成测试，可以建议优化器把顺序扫描看得更加昂贵。可以在会话中使用下面的指令让操作更加昂贵：

```
SET enable_seqscan TO off;
```

这样索引就可以按预期工作了：

```
test=# explain SELECT * FROM t sva WHERE sva = '0000112273';
                                  QUERY PLAN
------------------------------------------------------------------
 Index Only Scan using idx_special on t_sva (cost=0.13..8.14 rows=1
width=32)
    Index Cond: (sva = '0000112273'::text)
(2 rows)

test=# SELECT * FROM t sva;
sva
------------
2345010477
1118090878
(2 rows)
```

3.5 理解 PostgreSQL 索引类型

到目前为止只讨论了二叉树。不过，在很多情况下 B-树是不够的。为什么会有这样的情况？正如在本章中讨论的，B-树本质上是基于排序的。使用 B-树可以处理操作符<、<=、=、>=和>。问题是，并非所有的数据类型都可以以一种有用的方式排序。想象一个多边形，如何能把这些对象以有用的方式排序？当然，可以通过覆盖面积、周长等来排序，但是这无法让用户使用几何搜索真正找到它们。

这个问题的解决方案就是提供更多的索引类型。每一种索引将服务于一种特殊的目的并且完成所需要的工作。PostgreSQL 中有下列索引类型可用（从 PostgreSQL 9.6 开始）：

```
test=# SELECT * FROM pg_am;
 amname |  amhandler   | amtype
```

```
--------+-------------+--------
 btree  | bthandler   | i
 hash   | hashhandler | i
 GiST   | GiSThandler | i
 gin    | ginhandler  | i
 spGiST | spghandler  | i
 brin   | brinhandler | i
(6 rows)
```

总共有 6 种类型的索引。B-树已经被详细讨论过，但其他几种索引类型适合什么？下面的小节将会介绍 PostgreSQL 中可用的每一种索引类型的目的。

注意还有一些扩展可以被用在这里的索引类型之上。在网上还有一些额外的索引类型可用，如 rum、vodka 和未来的 cognac。

3.5.1　Hash 索引

Hash 索引已经存在了很多年。其思想是把输入值进行哈希并且将结果存起来用于后面的查找。使用 Hash 索引的确是有意义的。不过，在 PostgreSQL 中并不建议使用它们。原因是 Hash 索引在并发情况表现不好，并且不支持 PostgreSQL 事务日志。简而言之，在可以预见的未来都不应该在现实世界的系统中使用它们。

3.5.2　GiST 索引

通用搜索树（GiST）索引是一种非常重要的索引类型，因为它们应用于很多不同的东西。GiST 索引可以被用来实现 R-树行为，它甚至也可以作为 B-树来用。不过，不推荐把 GiST 滥用为 B-树索引。

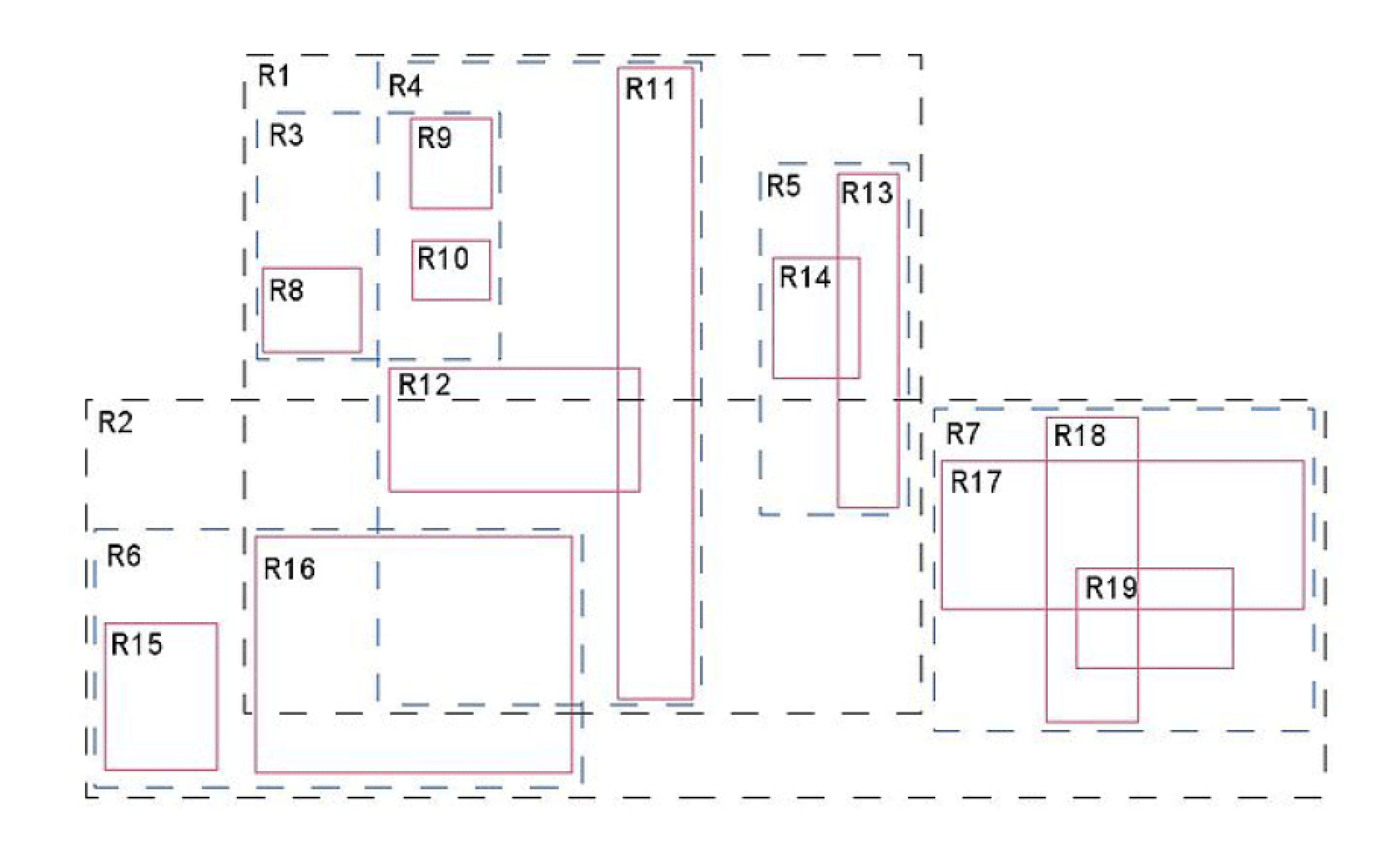
GiST 的典型用例有：

- 范围类型。
- 几何索引（例如，用于非常流行的 PostGIS 扩展）。
- 模糊搜索。

1. 理解 GiST 如何工作

对于很多人来说，GiST 仍然是一个黑盒子。因此，笔者决定增加一节来描述 GiST 的内部是如何工作的。

考虑图 3.1：



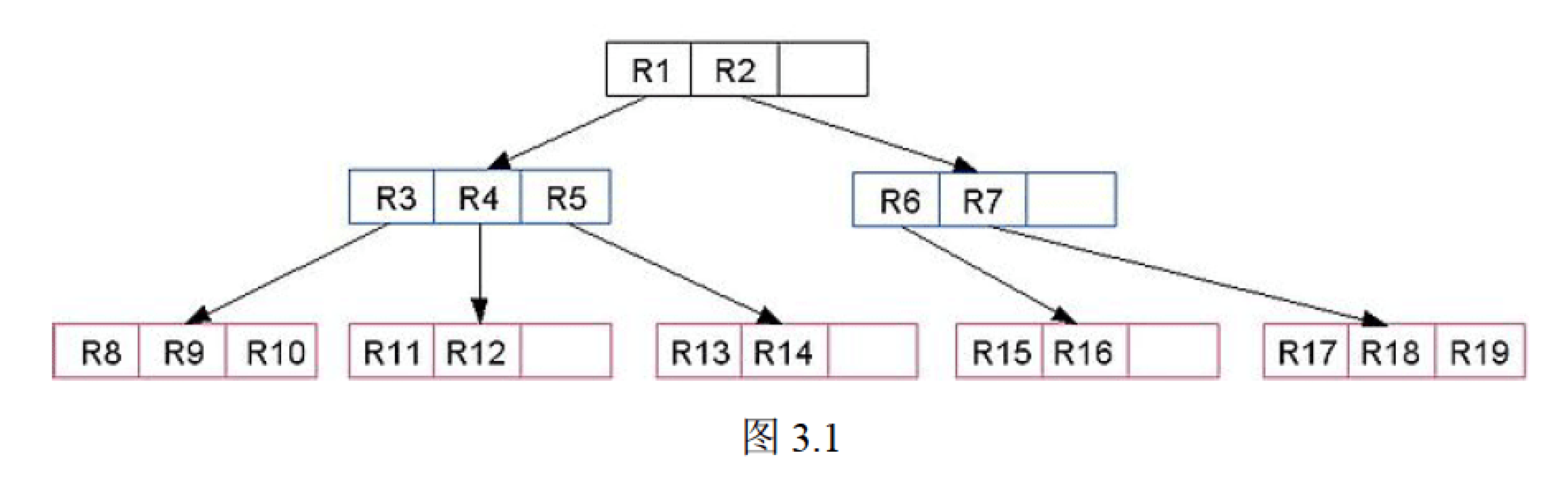


图 3.1

来源：http://leopard.in.ua/assets/images/postgresql/pg_indexes/pg_indexes2.jpg

看看这一棵树，读者将看到 R1 和 R2 在顶部。R1 和 R2 是包含所有其他东西的外包盒。R3、R4 和 R5 被 R1 所包含。R8、R9 和 R10 被 R3 所包含等。因此一个 GiST 索引是层次化组织的。读者在图 3.1 中看到的是某些在内建 B-树中没有的操作。那些操作是重叠、在左边、在右边等。一棵 GiST 树的布局对于几何索引非常理想。

2. 扩展 GiST

当然，也可以设计用户自己的操作符类。支持下列策略，如表 3-3 所示。

表 3-3

操　　作	策　略　号
Strictly left of	1
Does not extend to right of	2
Overlaps	3

续表

操　作	策　略　号
Does not extend to left of	4
Strictly right of	5
Same	6
Contains	7
Contained by	8
Does not extend above	9
Strictly below	10
Strictly above	11
Does not extend below	12

如果想要为 GiST 编写操作符类，就必须提供一些支持函数。在 B-树的情况下，只需要一个 same 函数，而 GiST 索引提供更多函数，如表 3-4 所示。

表 3-4

函　数	描　述	支持函数编号
consistent	该函数判断一个键是否满足查询限定词。在内部会查找并检查策略	1
union	计算一个键集合的联合。在数字值的情况下，就是计算较高和较低的值或者一个范围。它对于几何结构尤其重要	2
compress	计算一个键或者值的压缩表示	3
decompress	这是 compress 函数的逆函数	4
penalty	在插入过程中，插入到树中的代价将被计算出来。这个代价决定新的项会被放在树中的什么位置。因此，一个好的 penalty 函数是索引总体性能好坏的关键	5
picksplit	判断在页面分裂时从哪里开始移动项。一些项必须留在旧页面中，而其他项将会去到被创建的新页面中。一个好的 picksplit 函数是好的索引性能的根本	6
equal	equal 函数类似于已经在 B-树中见过的 same 函数	7
distance	计算一个键和查询值之间的距离（一个数字）。distance 函数是可选的，只有在支持 KNN 搜索时才需要	8
fetch	判断一个压缩键的原始表示。PostgreSQL 近期版本支持的只用索引扫描需要用这个函数来处理	9

GiST 索引的操作符类通常都用 C 来实现。如果想找一个好例子，建议去源代码的 contrib 目录中看一看 btree_GiST 模块。它展示了如何使用 GiST 索引标准数据类型，并且是一个好的灵感来源。

3.5.3　GIN 索引

通用倒排（GIN）索引是一种好的索引文本的方式。假定用户要索引一百万个文本文档，一个特定的词可能会出现几百万次。在一棵普通的 B-树中，这意味着键会被存储几百万次。但在 GIN 索引中不是这样。每一个键（或者词）只被存储一次并且会被附加一个文档列表。键以一棵标准的 B-树组织。每一个项将有一个文档列表，它指向表中具有这个相同键的所有项。一个 GIN 索引会非常小并且紧凑。不过，它缺少 B-树中一种重要的特性——排序数据。在 GIN 中，与一个特定键相关的项指针列表会按照行在表中的位置而排序，而不能采用一种随意的规则。

- 扩展 GIN

就和任何其他索引一样，GIN 可以被扩展。有下列策略可用，如表 3-5 所示。

表 3-5

操　　作	策　略　号
Overlap	1
Contains	2
Is contained by	3
Equal	4

在此之上，有下列支持函数可用，如表 3-6 所示。

表 3-6

函　　数	描　　述	支持函数编号
compare	compare 函数类似于已经在 B-树中见过的 same 函数。如果两个键被比较，它会返回：-1（小于）、0（等于）或者 1（大于）	1
extractValue	从一个要被索引的之中抽取键。一个值可能有很多键。例如，一个文本值可以由多于一个词组成	2
extractQuery	从一个查询条件中抽取键	3
consistent	检查一个值是否匹配一个查询条件	4
comparePartial	比较一个来自查询的部分键和一个来自索引的键。返回-1、0 或者 1（类似于 B-树支持的 same 函数）	5
triConsistent	判断一个值是否匹配一个查询条件（三元变体）。如果 consistent 函数存在，triConsistent 就是可选的	6

如果想要一个如何扩展 GIN 的例子，考虑看看在 PostgreSQL 源代码 contrib 目录中的

btree_gin 模块。它是一个有价值的信息源，也是一种开始自定义实现的好方法。

如果读者对全文搜索感兴趣，本章稍后还将提供更多信息。

3.5.4　SP-GiST 索引

空间划分 GiST（SP-GiST）主要被设计为在内存中使用。其原因是一个存储在磁盘上的 SP-GiST 需要相当高的磁盘命中才能工作。磁盘命中是比只在 RAM 沿着一些指针查找更加昂贵的方式。

美妙的事情是 SP-GiST 可以被用来实现多种类型的树，例如四叉树、k-d 树和 radix 树。

SP-GiST 提供了表 3-7 所示的策略。

表 3-7

操　　作	策　略　号
Strictly left of	1
Strictly right of	5
Same	6
Contained by	8
Strictly below	10
Strictly above	11

要为 SP-GiST 编写自己的操作符类，必须提供如表 3-8 所示的一些函数。

表 3-8

函　　数	描　　述	支持函数编号
config	提供使用的操作符类的信息	1
choose	确定如何把一个新值插入一个内部元组中	2
picksplit	确定如何划分/分裂一组值	3
inner_consistent	判断对于一个查询需要搜索哪些子分区	4
leaf_consistent	判断键是否满足查询限定词	5

3.5.5　BRIN 索引

块范围索引（BRIN）有很多实际用途。到目前为止，讨论过的所有索引都需要很多磁盘空间。尽管在缩小 GIN 等索引方面已经做了很多工作，但它们仍然需要很大的空间，因为每一个项都需要一个索引指针。因此如果有一千万个项，就需要有一千万个索

引指针。空间是 BRIN 索引主要关心的问题。BRIN 索引并不为每一个元组保存一个索引项，而是存储数据的 128（默认）个块（1MB）中的最小和最大值。因此这种索引很小，但是一种有损索引。扫描这种索引将会返回比要求的更多的数据。PostgreSQL 必须在后续的步骤中过滤掉那些额外的行。

下面的例子展示了一个 BRIN 索引到底有多小：

```
test=# CREATE INDEX idx_brin ON t_test USING brin(id);
CREATE INDEX
test=# \di+ idx_brin
                             List of relations
 Schema |    Name    | Type  | Owner | Table   | Size
--------+----------+-------+-------+--------+---------------------
 public | idx_brin | index | hs     | t_test | 48 KB
(1 row)
```

在笔者的例子中，BRIN 索引比一个标准的 B-树小 2000 倍。但问题自然就产生了：为什么我们不总是使用 BRIN 索引呢？要回答这一类问题，了解 BRIN 的布局很重要，在 BRIN 中存储了 1MB 数据的最小值和最大值。如果数据是排序的（关联度高），BRIN 是相当有效的，因为我们可以取出 1MB 数据并扫描它即可。但是，如果数据是乱的会怎样？在这种情况下，BRIN 将无法排除数据块，因为很可能有接近于总体最高值和最低值的数据在这 1MB 中。因此，BRIN 更多的是用于高关联的数据。实际上，在数据仓库应用中高度关联的数据是很有可能的。例如，通常每天装载数据，所以日期可能会高度关联。

- 扩展 BRIN 索引

BRIN 支持和 B-树相同的策略，因此它需要相同的操作符集合。代码可以很好地被重用，如表 3-9 所示。

表 3-9

操　作	策　略　号
Less than	1
Less than or equal	2
Equal	3
Greater than or equal	4
Greater than	5

BRIN 所需的支持函数如表 3-10 所示。

表 3-10

函　　数	描　　述	支持函数编号
opcInfo	提供关于被索引列的内部信息	1
add_value	向一个现有的摘要元组增加一项	2
consistent	检查一个值是否匹配一个条件	3
union	计算两个摘要项的联合（最小/最大值）	4

3.5.6　增加额外索引

从 PostgreSQL 9.6 开始，有一种容易的方式以扩展的形式发布全新的索引类型。因为如果 PostgreSQL 提供的那些索引类型不够用，就可以增加几种额外的索引类型来准确地实现用户的目的。增加索引类型的指令是 CREATE ACCESS METHOD：

```
test=# \h CREATE ACCESS METHOD
Command:     CREATE ACCESS METHOD
Description: define a new access method
Syntax:
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
```

不用太担心这个命令——万一用户要部署自己的索引类型，它会以现成的扩展形式出现。

其中一个扩展实现了布隆过滤器。布隆过滤器是概率数据结构，它们有时会返回过多的行，但是从不会返回过少的行。因此，布隆过滤器是一种预先过滤数据的好方法。

布隆过滤器是怎样工作的呢？一个布隆过滤器被定义在几个列上。基于输入值会计算出一个位掩码，位掩码会接着与查询进行比较。布隆过滤器的优势是可以想索引多少列就索引多少列。而其劣势是整个布隆过滤器都必须被读取。当然，布隆过滤器比其底层的数据要小很多，因此在很多情况下还是很划算的。

要使用布隆过滤器，只需要激活该扩展，它本身就是 PostgreSQL 的 contrib 包的一部分：

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

如前所述，布隆过滤器背后的思想是允许索引需要的任意多列。在很多实际应用中，挑战在于索引很多列却不知道用户在运行时真正需要哪些组合。在一个大型表的情况下，完全不可能在 80 个或者更多个域上创建一个标准的 B-树索引。这时候布隆过滤器

可能是另一种选择：

```
test=# CREATE TABLE t_bloom (x1 int, x2 int, x3 int, x4 int, x5 int, x6
int, x7 int);
CREATE TABLE
```

创建索引很容易：

```
test=# CREATE INDEX idx_bloom ON t_bloom (x1, x2, x3, x4, x5, x6, x7);
CREATE INDEX
```

如果关闭顺序扫描，该索引就可以发挥作用：

```
test=# explain SELECT * FROM t_bloom WHERE x5 = 9 AND x3 = 7;
                            QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on t_bloom (cost=18.50..22.52 rows=1 width=28)
   Recheck Cond: ((x3 = 7) AND (x5 = 9))
   -> Bitmap Index Scan on idx bloom (cost=0.00..18.50 rows=1 width=0)
        Index Cond: ((x3 = 7) AND (x5 = 9))
```

注意笔者查询的是随机的几个列的组合，它们和索引中的实际顺序无关。布隆过滤器仍能有所帮助。

如果对布隆过滤器感兴趣，可以参考网站：https://en.wikipedia.org/wiki/Bloom_filter。

3.6 用模糊搜索实现更好的回答

当今的用户并不是只需要执行精确搜索。现代的网站已经让用户习惯了总是能够得到一个结果，而不管用户输入什么。如果在 Google 上搜索，即便用户的输入是错误的、错字连篇或者毫无意义，总是会有一个回答。人们总是期待好的结果而不管输入数据是怎样的。

3.6.1 利用 pg_trgm

要用 PostgreSQL 作模糊搜索，可以加入 pg_trgm 扩展。要激活该扩展，只需要运行下面的指令：

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

pg_trgm 扩展相当强大，为了展示它的能力，笔者已经编好了一些由奥地利 2354 个

村庄和城市的名字构成的示例数据。

我们的示例数据可以存储在一张简单表中：

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE
```

笔者的公司网站有全部的数据，PostgreSQL 可以直接装载该数据：

```
test=# COPY t_location FROM PROGRAM 'curl www.cybertec.at/secret/orte.txt';
COPY 2354
```



注意必须安装 curl（一种获取数据的命令行工具）。如果没有这个工具，请正常下载该文件，并从本地文件系统中导入它。

一旦数据被装载完，就可以查询该表的内容：

```
test=# SELECT * FROM t_location LIMIT 4;
              name
--------------------------------
 Eisenstadt
 Rust
 Breitenbrunn am Neusiedler See
 Donnerskirchen
(4 rows)
```

如果德语不是你的母语，就不太可能拼写那些位置的名称而不出现严重的错误。幸运地是，pg_trgm 可以帮助你：

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

pg_trgm 为我们提供了一种距离操作符，它能计算两个字符串之间的距离：

```
test=# SELECT 'abcde' <-> 'abdeacb';
 ?column?
----------
 0.833333
(1 row)
```

这种距离是 0 和 1 之间的一个数字。数字越小，两个字符串就越相似。

那是怎样做到的呢？trigram 会拿到一个字符串并且把它解剖成三字母序列：

```
test=# SELECT show_trgm('abcdef');
              show_trgm
-------------------------------------
```

```
 {" a"," ab",abc,bcd,cde,def,"ef "}
(1 row)
```

这些序列接着将被用来算出刚才看到的距离。当然，这个距离操作符可以被用在一个查询内来寻找最接近的匹配：

```
test=# SELECT * FROM t_location ORDER BY name <-> 'Kramertneusiedel' LIMIT
3;
       name
-----------------
 Gramatneusiedl
 Klein-Neusiedl
 Potzneusiedl
(3 rows)
```

Gramatneusiedl 与 Kramertneusiedel 相当接近。两者听起来很相似并且使用一个 K 来代替 G 是一种很常见的错误。在 Google 上，有时会看到 **Did you mean...**。这很可能就是 Google 在使用 n-gram。

PostgreSQL 可以用 GiST 在文本上通过 trigram 建立索引：

```
test=# CREATE INDEX idx trgm ON t location USING GiST(name GiST trgm ops);
CREATE INDEX
```

pg_trgm 为我们提供了 GiST_trgm_ops 操作符类来做相似性匹配。下面的列表展示了该索引可以发挥预期的作用：

```
test=# explain SELECT * FROM t_location ORDER BY name <->
'Kramertneusiedel' LIMIT 5;
                              QUERY PLAN
------------------------------------------------------------------
 Limit (cost=0.14..0.58 rows=5 width=17)
   -> Index Scan using idx trgm on t location
        (cost=0.14..207.22 rows=2354 width=17)
        Order By: (name <-> 'Kramertneusiedel'::text)
(3 rows)
```

3.6.2 加速 LIKE 查询

LIKE 查询肯定会导致一些当今用户所见过的最糟糕的性能问题。在大部分数据库系统中，LIKE 的速度相当慢并且要求一次顺序扫描。除此之外，最终用户很快会发现一个模糊搜索大部分情况下将返回比精确查询更好的结果。因此，如果足够频繁地在一个大

型表上调用一种单一类型的 LIKE 查询，常常会削弱整个数据库服务器的性能。

幸运地是，PostgreSQL 为这一问题提供了解决方案，并且该解决方案正好已经被安装：

```
test=# explain SELECT * FROM t_location WHERE name LIKE '%neusi%';
                              QUERY PLAN
-----------------------------------------------------------------
 Bitmap Heap Scan on t location
   (cost=4.33..19.05 rows=24 width=13)
   Recheck Cond: (name ~~ '%neusi%'::text)
   -> Bitmap Index Scan on idx trgm (cost=0.00..4.32 rows=24 width=0)
        Index Cond: (name ~~ '%neusi%'::text)
(4 rows)
```

3.6.1 节中部署的 trigram 索引也适合于加速 LIKE 查询。注意%符号可以被用在搜索串中的任意位置。这是相对于标准 B-树的一个主要优势，标准 B-树只能加速通配符在查询末端的情况。

3.6.3　处理正则表达式

不过，这还不是 trigram 的全部作用。trigram 索引甚至能够加速简单的正则表达式。下面的例子展示了这种功效：

```
test=# SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';
     name
--------------
Bruckneudorf
(1 row)

test=# explain SELECT * FROM t location WHERE name ~ '[A-C].*neu.*';
                                QUERY PLAN
-----------------------------------------------------------------
 Index Scan using idx_trgm on t_location (cost=0.14..8.16
    rows=1 width=13)
    Index Cond: (name ~ '[A-C].*neu.*'::text)
(2 rows)
```

PostgreSQL 将检查该正则表达式并且使用该索引来回答问题。



在内部，PostgreSQL 可以把该正则表达式转换成一个图并且相应地遍历该索引。

3.7　理解全文搜索-FTS

如果正在查找名称或者简单的字符串，用户通常需要查询一个域的整个内容。在 FTS 中不是这样。全文搜索的目的是在一个文本中查找词或者词组。因此，FTS 更大程度上是一种包含操作，因为基本上不会用它来查找一个准确的字符串。

在 PostgreSQL 中，可以使用 GIN 索引来做 FTS。其思想是解剖一个文本，抽出有价值的词位并且索引这些元素而不是底层的文本。为了让搜索更加成功，那些词会被预处理。

这里是一个例子：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars');
                              to_tsvector
-----------------------------------------------------------------
 'car':2,6,14 'even':10 'mani':13 'mind':11 'want':4 'would':8
(1 row)
```

这个例子展示了一个简单的句子。to_tsvector 函数将获取该字符串，应用英语规则并且执行一个词干提取处理。基于配置（english），PostgreSQL 将解析该字符串，丢掉停用词并且提取单词的词干。例如，car 和 cars 将被转换成 car。注意这与查找词干无关。对于 many，PostgreSQL 将简单地通过应用英语语言中工作得很好的标准规则把该字符串转换成 mani。

注意，to_tsvector 函数的输出是与语言高度相关的。如果告诉 PostgreSQL 把字符串当作 Dutch 对待，结果将完全不同：

```
test=# SELECT to tsvector('dutch', 'A car, I want a car. I would not even
mind having many cars');
                             to tsvector
-------------------------------------------------------------------
 'a':1,5 'car':2,6,14 'even':10 'having':12 'i':3,7 'many':13
   'mind':11 'not':9 'would':8
(1 row)
```

要搞明白支持哪些配置，考虑运行下面的查询：

```
SELECT cfgname FROM pg_ts_config;
```

3.7.1　比较字符串

在简单地看过了词干提取处理之后，让我们看看怎样把经过词干提取处理的文本与

一个用户查询相比较。下面的代码片段会检查 wanted：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to_tsquery('english', 'wanted');
 ?column?
----------
 t
(1 row)
```

注意，wanted 实际上并未出现在原始文本中。但是 PostgreSQL 仍然返回真。原因是 want 和 wanted 都被转换成相同的词位，因此结果是真。实际上，这是很有意义的。想象一下用户正在 Google 上查找 car。如果能找到售卖 cars 的页面，这是相当不错的。因此查找公共词位是一种聪明的想法。

有时候，人们不只是查找单一的词而是想找到一组词。如下一个例子所示，to_tsquery 可以做到这一点：

```
test=# SELECT to tsvector('english', 'A car, I want a car. I would not even
mind having many cars') @@ to tsquery('english', 'wanted & bmw');
 ?column?
----------
 f
(1 row)
```

这种情况会返回假，因为在输入字符串中找不到 bmw。在 to_tsquery 函数中，“&”表示与而“|”表示或。因此我们可以轻易地构建复杂的搜索串。

3.7.2　定义 GIN 索引

如果想在一列或者一组列上应用文本搜索，基本上有两种选择：

- 创建一个使用 GIN 的函数索引。
- 增加一个类型为 tsvector 的列和一个触发器来保持文本列和 tsvector 列同步。

在本节中，两种选项都会被介绍。为了展示如何使用这些选项，笔者创建了一些示例数据：

```
test=# CREATE TABLE t fts AS SELECT comment FROM pg available extensions;
SELECT 43
```

直接在列上建立函数索引肯定是一种较慢但是空间效率更高的方法：

```
test=# CREATE INDEX idx fts func ON t fts USING gin(to tsvector('english',
comment));
CREATE INDEX
```

在函数上部署一个索引很容易，但是可能会带来一些负荷。增加一个物化列需要更多空间，但是会得到更好的运行时行为：

```
test=# ALTER TABLE t_fts ADD COLUMN ts tsvector;
ALTER TABLE
```

唯一的麻烦是：如何保持这个列同步？答案是用一个触发器：

```
test=# CREATE TRIGGER tsvectorupdate
BEFORE INSERT OR UPDATE ON t fts
FOR EACH ROW
EXECUTE PROCEDURE
tsvector_update_trigger(somename, 'pg_catalog.english', "comment");
```

幸运的是，PostgreSQL 已经提供了一个可以被用作触发器来同步 tsvector 列的 C 函数。只需要向该函数传入一个名称、想要的语言以及一组列就可以了。该触发器函数将做好一切所需要的工作。注意一个触发器将总是和造成修改的语句处于同一个事务中，因此不会有造成不一致的风险。

3.7.3　调试用户的搜索

有时并不太清楚为什么一个查询能匹配一个给定的搜索串。为了调试查询，PostgreSQL 提供了 ts_debug 函数。从用户的角度来看，它可以像 to_tsvector 一样使用。它透露了很多有关 FTS 内部工作的信息：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM ts debug('english', 'go to
www.postgresql-support.de');
-[ RECORD 1 ]+-----------------------------
alias        | asciiword
description  | Word, all ASCII
token        | go
dictionaries | {english_stem}
dictionary   | english stem
lexemes      | {go}
-[ RECORD 2 ]+-----------------------------
alias        | blank
description  | Space symbols
token        |
dictionaries | {}
```

```
dictionary    |
lexemes       |
-[ RECORD 3 ]+---------------------------
alias         | asciiword
description   | Word, all ASCII
token         | to
dictionaries  | {english stem}
dictionary    | english_stem
lexemes       | {}
-[ RECORD 4 ]+---------------------------
alias         | blank
description   | Space symbols
token         |
dictionaries  | {}
dictionary    |
lexemes       |
-[ RECORD 5 ]+---------------------------
alias         | host
description   | Host
token         | www.postgresql-support.de
dictionaries  | {simple}
dictionary    | simple
lexemes       | {www.postgresql-support.de}
```

ts_debug 将列出每一个找到的记号并且显示有关该记号的信息。用户会看到解析器找到了哪些记号、用到的词典以及对象的类型。在笔者的例子中，解析器找到了空格、单词和主机。用户可能还会看到数字、email 地址和很多其他的东西。根据字符串的类型，PostgreSQL 将以不同的方式来处理。例如，对主机名和 email 地址提取词干绝对是没有意义的。

3.7.4　收集词统计信息

全文搜索可以处理很多数据。为了让最终用户更深入地查看他们的文本，PostgreSQL 提供了 pg_stat 函数，它会返回一个词的列表：

```
SELECT * FROM ts_stat('SELECT to_tsvector(''english'', comment) FROM
pg_available_extensions') ORDER BY 2 DESC LIMIT 3;
   word    | ndoc | nentry
----------+------+--------
 function | 10   | 10
 data     | 10   | 10
```

```
 type        | 7      | 7
(3 rows)
```

word 列包含经过提取词干处理的词，ndoc 告诉我们一个特定词出现的文档数。nentry 表示一个词总共被找到多少次。

3.7.5　利用排除操作符

到目前为止，索引已经被用来加速操作以及确保唯一性。不过，几年以前一些人想出了索引的更多用途。正如读者在本章中已经见过的，GiST 支持相交、重叠、包含等很多操作。那么为什么不把这些操作用来管理数据完整性呢？

这里是一个例子：

```
test=# CREATE EXTENSION btree_gist;
test=# CREATE TABLE t reservation (
        room int,
        from to tsrange,
        EXCLUDE USING GiST (room with =,
                             from_to with &&)
);
CREATE TABLE
```

EXCLUDE USING GiST 子句定义额外的约束。如果客户正在售卖房间，他/她可能想允许不同的房间在同一时间被预订。不过，客户不会想在同一段时期内把同一间房卖出两次。在笔者的例子中，EXCLUDE 子句表达的意思是：如果房间相等，from_to with 中的数据就不能重叠（&&）。

下面的两行将不会违背约束：

```
test=# INSERT INTO t reservation VALUES (10, '["2017-01-01",
"2017-03-03"]');
INSERT 0 1
test=# INSERT INTO t_reservation VALUES (13, '["2017-01-01",
"2017-03-03"]');
INSERT 0 1
```

不过，接下来的 INSERT 将会导致违背，因为出现了数据重叠：

```
test=# INSERT INTO t_reservation VALUES (13, '["2017-02-02",
"2017-08-14"]');
ERROR: conflicting key value violates exclusion constraint
"t reservation room from to excl"
DETAIL: Key (room, from_to)=(13, ["2017-02-02 00:00:00","2017-08-14
```

```
00:00:00"]) conflicts with existing key (room, from_to)=(13, ["2017-01-
0100:00:00","2017-03-03 00:00:00"]).
```

排除操作符的使用非常有用，它可以为用户提供非常高级的方法来处理完整性。

3.8　总　　结

本章全都是有关索引的内容。我们已经学到了何时 PostgreSQL 将决定使用索引以及有哪些类型的索引存在。除了仅仅使用索引，还可以实现我们自己的策略，用自定义的操作符和索引策略来加速应用。

对于那些追求极致的人，PostgreSQL 还提供了自定义访问方法。

第 4 章全部都是有关高级 SQL 的内容。很多人并没有意识到 SQL 的真正能力，因此笔者将为人们展示一些高效的、更加高级的 SQL。

第 4 章　处理高级 SQL

在第 3 章中，读者已经学到了索引以及 PostgreSQL 运行自定义索引代码加速查询的能力。在本章中，读者将学到有关高级 SQL 的内容。本书的大部分读者都有一些使用 SQL 的经验，但是，经验表明，本书中介绍的大部分高级特性并非广为人知。因此在这里介绍这些特性有助于人们更快、更有效地达成他们的目标。

本章的主题包括：

- 分组集。
- 排序集。
- 假想聚集。
- 窗口函数及分析。

在本章结束时，读者将能够理解和使用高级 SQL。

4.1　引入分组集

每一个 SQL 的高级用户应该都很熟悉 GROUP BY 和 HAVING 子句。但读者是否也知道 CUBE、ROLLUP 以及 GROUPING SETS？如果不知道，本章将值得一读。

4.1.1　装载一些案例数据

为了让读者在本章中获得更愉快的体验，笔者已经编写了一些案例数据，它们来自于 BP 能源报告：http://www.bp.com/en/global/corporate/energy-economics/statistical-review-of- world-energy.html。

这里是将要用到的数据结构：

```
test=# CREATE TABLE t_oil (
  region          text,
  country         text,
  year            int,
  production      int,
  consumption     int
);
CREATE TABLE
```

用 curl 可以直接从我们的网站下载这一测试数据：

```
test=# COPY t_oil FROM PROGRAM ' curl www.cybertec.at/secret/oil_ext.txt ';
COPY 644
```

和第 3 章一样，该文件可以在导入之前先下载它。在一些操作系统上，默认没有安装 curl，因此对于很多人来说先下载该文件可能是更容易的选项。

这份数据中包含世界上两个地区共计 14 个国家在 1965 年到 2010 年间的数据：

```
test=# SELECT region, avg(production) FROM t_oil GROUP BY region;
     region     |          avg
---------------+-----------------------
 Middle East    | 1992.6036866359447005
 North America | 4541.3623188405797101
(2 rows)
```

4.1.2　应用分组集

GROUP BY 子句将把很多行转变成每一组一行。不过，如果是在做现实生活中的报表，用户可能还对总体的平均值感兴趣。因此有时可能需要一个附加的行。

可以这样来做：

```
test=# SELECT region, avg(production) FROM t oil GROUP BY ROLLUP (region);
     region     |          avg
---------------+-----------------------
 Middle East    | 1992.6036866359447005
 North America | 4541.3623188405797101
                | 2607.5139860139860140
(3 rows)
```

ROLLUP 将会注入一个附加的行，其中包含总体的平均值。如果用户正在做报告，它就非常像所需要的总结行。无须运行两个查询，PostgreSQL 可以仅用一个查询就提供上述这样的数据。

当然，这种操作在通过不止一列分组时也能使用：

```
test=# SELECT region, country, avg(production) FROM t oil WHERE country
IN ('USA', 'Canada', 'Iran', 'Oman') GROUP BY ROLLUP (region, country);
     region     | country |          avg
---------------+---------+-----------------------
 Middle East    | Iran     | 3631.6956521739130435
 Middle East    | Oman     | 586.4545454545454545
```

```
 Middle East   |         | 2142.9111111111111111
 North America | Canada  | 2123.2173913043478261
 North America | USA     | 9141.3478260869565217
 North America |         | 5632.2826086956521739
               |         | 3906.7692307692307692
(7 rows)
```

在这个例子中，PostgreSQL 将在结果集中注入 3 行。一行针对 Middle East，一行针对 North America。此外还将有一行是总体平均值。如果用户正在构建一个 Web 应用，那么当前的结果就很理想，因为可以很容易地构建一个 GUI 通过过滤掉空值来钻入结果集中。

如果立即想要显示一个结果，ROLLUP 就很不错。笔者总是用它来把最终结果显示给最终用户。不过，如果用户正在做报告，他/她可能想要预先计算更多数据来确保更好的灵活性。CUBE 关键词就是用户所需要的东西：

```
test=# SELECT region, country, avg(production) FROM t oil WHERE country
IN ('USA', 'Canada', 'Iran', 'Oman') GROUP BY CUBE (region, country);
     region    | country |          avg
---------------+---------+-----------------------
 Middle East   | Iran    | 3631.6956521739130435
 Middle East   | Oman    | 586.4545454545454545
 Middle East   |         | 2142.9111111111111111
 North America | Canada  | 2123.2173913043478261
 North America | USA     | 9141.3478260869565217
 North America |         | 5632.2826086956521739
               |         | 3906.7692307692307692
               | Canada  | 2123.2173913043478261
               | Iran    | 3631.6956521739130435
               | Oman    | 586.4545454545454545
               | USA     | 9141.3478260869565217
(11 rows)
```

注意甚至有更多的行被加入结果中。CUBE 创建的数据是：GROUP BY region, country + GROUP BY region + GROUP BY country + 总体均值。因此，其整体思想是同时抽取很多结果以及各个层面的聚集。结果立方体包含所有可能的分组组合。

ROLLUP 和 CUBE 实际只是在 GROUPING SETS 子句之上的便利特性。通过 GROUPING SETS 子句，读者可以明确地列出想要的聚集：

```
test=# SELECT region, country, avg(production)
FROM t_oil
WHERE country IN ('USA', 'Canada', 'Iran', 'Oman')
```

```
GROUP BY GROUPING SETS ( (), region, country);
    region      | country |          avg
----------------+---------+-----------------------
 Middle East    |         | 2142.9111111111111111
 North America  |         | 5632.2826086956521739
                |         | 3906.7692307692307692
                | Canada  | 2123.2173913043478261
                | Iran    | 3631.6956521739130435
                | Oman    | 586.4545454545454545
                | USA     | 9141.3478260869565217
(7 rows)
```

在这里，我们得到了 3 个分组集：总体均值、GROUP BY region 和 GROUP BY country。如果读者想要组合 region 和 country，可以使用(region, country)。

- 性能研究

分组集是一种强大的特性，它们有助于减少昂贵查询的数量。在内部，PostgreSQL 基本上会把分组集转成传统的 GroupAggregates 来实现。GroupAggregate 节点要求排序好的数据，因此要做好 PostgreSQL 可能会执行很多临时排序的准备：

```
test=# explain SELECT region, country, avg(production)
FROM   t_oil
WHERE  country IN ('USA', 'Canada', 'Iran', 'Oman')
GROUP BY GROUPING SETS ( (), region, country);
                            QUERY PLAN
---------------------------------------------------------------
 GroupAggregate  (cost=22.58..32.69 rows=34 width=52)
   Group Key: region
   Group Key: ()
   Sort Key: country
     Group Key: country
   -> Sort  (cost=22.58..23.04 rows=184 width=24)
         Sort Key: region
         -> Seq Scan on t oil
             (cost=0.00..15.66 rows=184 width=24)
             Filter: (country = ANY
                ('{USA,Canada,Iran,Oman}'::text[]))
(9 rows)
```

只有不涉及分组集的普通 GROUP BY 子句才支持哈希聚集。根据分组集的开发者（Atri Shama）所说（笔者就在写本章之前刚和他探讨过），对哈希的支持不值得增加。因

此看起来 PostgreSQL 已经有了一种有效的实现，虽然优化器在这方面拥有的选择比普通 GROUP BY 语句更少。

4.1.3　组合分组集和 FILTER 子句

在实际应用中，分组集可能会经常与 FILTER 子句组合在一起。其思想是使用 FILTER 子句运行部分聚集。

这里是一个例子：

```
test=# SELECT region,
avg(production) AS all,
avg(production) FILTER (WHERE year < 1990) AS old,
avg(production) FILTER (WHERE year >= 1990) AS new
FROM t_oil
GROUP BY ROLLUP (region);
     region    |       all       |       old       |       new
---------------+-----------------+-----------------+----------------
 Middle East   | 1992.603686635 | 1747.325892857 | 2254.233333333
 North America | 4541.362318840 | 4471.653333333 | 4624.349206349
               | 2607.513986013 | 2430.685618729 | 2801.183150183
(3 rows)
```

这个例子的思想是，并非所有的列都会使用相同的数据来聚集。FILTER 子句允许用户有选择地传递数据给那些聚集。在笔者的例子中，第二个聚集将只考虑 1990 年之前的数据，而第三个聚集[①]将关注较新的数据。



如果可以把条件移到 WHERE 子句中那就最好，因为从表中取得的数据当然是越少越好。只有当并非每个聚集都需要被 WHERE 子句留下来的数据时才应该使用 FILTER。

FILTER 对所有类型的聚集都有效并且提供了一种简单的方式对数据做 pivot 操作。

4.2　使用有序集

有序集是一种强大的特性，但在开发者社区中并未受到广泛的关注和了解。其想法实际上很简单：数据被正常分组，然后将每个分组中的数据按照给定条件排序，最后在这种排序好的数据上做计算。

[①] 原文是“第二个聚集”，应为笔误。

一个经典的例子是中位数计算。



中位数是中间的值。例如，如果一个人的收入是中位数，那么收入比他（她）高和低的人数是相等的。50%的人收入更高并且 50%的人收入更低。

一种得到中位数的方法是得到排序好的数据并且把 50%移入数据集中。这里有一个例子展示了 WITHIN GROUP 子句将会让 PostgreSQL 做什么：

```
test=#  SELECT  region,  percentile disc(0.5)  WITHIN  GROUP  (ORDER  BY
production) FROM          t oil GROUP BY 1;
     region     | percentile disc
---------------+-----------------
 Middle East    | 1082
 North America | 3054
(2 rows)
```

percentile_disc 将跳过该组的 50%并且返回想要的值。注意，中位数可能会明显地偏离均值。在经济学中，中位数收入和收入均值之间的偏离甚至可能被用作社会平等的指示器。中位数相对于均值越高，则收入越不平等。为了提供更高的灵活性，ANSI 标准没有只提出一种中位数函数。相反，percentile_disc 允许用户使用任何介于 0 和 1 之间的值。

最妙的是用户甚至可以把有序集和分组集一起使用：

```
test=# SELECT region,
          percentile_disc(0.5) WITHIN GROUP (ORDER BY production)
FROM      t_oil
GROUP BY ROLLUP (1);
    region      | percentile_disc
---------------+-----------------
 Middle East    | 1082
 North America | 3054
                | 1696
(3 rows)
```

在这种情况下，PostgreSQL 又会在结果集中注入额外的行。

按照 ANSI SQL 标准所提出的，PostgreSQL 为用户提供了两种 percentile_函数。percentile_disc 将返回一个值，它是数据集中实际包含的值。percentile_cont 将在找不到精确匹配时插值。下面的例子展示了这一效果：

```
test=# SELECT
          percentile disc(0.62) WITHIN GROUP (ORDER BY id),
          percentile cont(0.62) WITHIN GROUP (ORDER BY id)
FROM      generate_series(1, 5) AS id;
```

```
 percentile_disc | percentile_cont
-----------------+-----------------
 4               | 3.48
(1 row)
```

4 是一个实际存在的值，而 3.48 是被插值的。

PostgreSQL 不仅提供了 percentile_函数，还有 mode 函数可用在分组中查找最频繁的值。在展示一个如何使用 mode 函数的例子之前，笔者编写了一个查询来展示更多有关该表内容的信息：

```
test=# SELECT production, count(*)
   FROM t_oil
   WHERE country = 'Other Middle East'
   GROUP BY production
   ORDER BY 2 DESC
   LIMIT 4;
 production | count
------------+-------
         50 |     5
         48 |     5
         52 |     5
         53 |     4
(4 rows)
```

3 个不同的值出现了正好 5 次。当然，mode 函数只能给出其中一个：

```
test=# SELECT country, mode() WITHIN GROUP (ORDER BY production)
   FROM t_oil
   WHERE country = 'Other Middle East'
   GROUP BY 1;
      country       | mode
--------------------+------
 Other Middle East |   48
(1 row)
```

这个例子返回了最频繁的值，但 SQL 不会告诉我们这个值实际出现得有多频繁。甚至可能该数字只出现了一次。

4.3　理解假想聚集

假想聚集很像标准的有序集。不过，它们能帮助回答一种不同类型的问题：如果一

个值在其中，那么结果会怎样？如你所见，这不是关于数据库内实际存在的值，而是有关一个特定值真实存在时的假想结果。

PostgreSQL 提供的唯一一种假想函数是 rank。它告诉我们：

```
test=# SELECT region,
              rank(9000) WITHIN GROUP
              (ORDER BY production DESC NULLS LAST)
FROM t oil
GROUP BY ROLLUP (1);
    region     | rank
---------------+------
 Middle East   |   21
 North America |   27
               |   47
(3 rows)
```

如果某地区日产 9000 桶，那将是北美地区第 27 好的年份以及中东地区第 21 好的年份。



笔者的例子使用了 NULLS LAST。在数据被排序时，空值通常都排在最后面。不过，即使排序顺序反过来，空值应该仍然在列表的末尾，NULLS LAST 就能确保这一点。

4.4　利用窗口函数和分析

在讨论了有序集之后，是时候看看窗口函数了。聚集遵循一种相当简单的原则：取得很多行并且把它们转变成较少的、聚集起来的行。窗口函数则不同，它把当前行与分组中的所有行进行对比。不过返回的行数没有变化。

下面是一个例子：

```
test=# SELECT avg(production) FROM t oil;
    avg
-----------
 2607.5139
(1 row)

test=#  SELECT  country,  year,  production,  consumption,  avg(production)
OVER()
FROM t oil
LIMIT 4;
```

```
 country | year | production | consumption |    avg
---------+------+------------+-------------+----------
 USA     | 1965 | 9014       | 11522       | 2607.5139
 USA     | 1966 | 9579       | 12100       | 2607.5139
 USA     | 1967 | 10219      | 12567       | 2607.5139
 USA     | 1968 | 10600      | 13405       | 2607.5139
(4 rows)
```

在我们的数据中，平均产量大约是每日 2600000 桶。这个查询的目标是把这个值增加为一列，这样就很容易把当前行与总体均值做对比。

一定记住 OVER 子句是必不可少的，没有它 PostgreSQL 无法处理该查询：

```
test=# SELECT country, year, production, consumption, avg(production)
FROM t_oil;
ERROR: column "t oil.country" must appear in the GROUP BY clause or be
used in an aggregate function
LINE 1: SELECT country, year, production, consumption, avg(productio...
```

这确实是有意义的，因为均值必须被精确定义，数据库引擎不能接受一个可能是猜测出来的值。



其他数据库引擎可能会接受不带 OVER 甚至不带 GROUP BY 子句的聚集函数。不过，从逻辑的角度来看这是错误的并且是对 SQL 的一种违背。

4.4.1　划分数据

到目前为止，相同的结果也都可以很轻易地用一个子查询实现。不过，如果用户想要的比总体均值更多，子查询将把查询变成噩梦。假定用户不想要总体均值而是正在处理的国家的均值，那就需要一个 PARTITION BY 子句：

```
test=# SELECT country, year, production, consumption, avg(production)
OVER (PARTITION BY country) FROM t_oil;
     country      | year | production | consumption |    avg
-----------------+------+------------+-------------+-----------
Canada           | 1965 |        920 |        1108 | 2123.2173
Canada           | 2010 |       3332 |        2316 | 2123.2173
Canada           | 2009 |       3202 |        2190 | 2123.2173
...
Iran             | 1966 |       2132 |         148 | 3631.6956
Iran             | 2010 |       4352 |        1874 | 3631.6956
Iran             | 2009 |       4249 |        2012 | 3631.6956
...
```

这里的重点是每一个国家都被分配了一个该国家的均值。OVER 子句定义我们正在查看的窗口，在这个例子中窗口是行所属的国家。换句话说，该查询返回行与其所在国家的所有行的对比。



year 列没有被排序。该查询并没有包含一个明确的排序顺序，因此数据可能会以随机顺序返回。记住，除非用户明确地说明想要的顺序，否则 SQL 并不承诺有序的输出。

基本上，PARTITION BY 子句可以接受任何表达式。通常大部分人会使用一个列来划分数据，例如：

```
test=# SELECT year, production,
   avg(production) OVER (PARTITION BY year < 1990)
FROM      t_oil
WHERE     country = 'Canada'
ORDER BY year;
 year | production |          avg
------+------------+-----------------------
 1965 |        920 | 1631.6000000000000000
 1966 |       1012 | 1631.6000000000000000
...
 1990 |       1967 | 2708.4761904761904762
 1991 |       1983 | 2708.4761904761904762
 1992 |       2065 | 2708.4761904761904762
...
```

这里的重点是数据划分采用表达式进行。year < 1990 可能返回两种值：true 和 false。根据一个年份所在的分组，它将会被分配 1990 年之前的均值或者 1990 年之后的均值，从这里可以看到 PostgreSQL 确实很灵活。使用函数来判断分组成员关系在实际应用中并不鲜见。

4.4.2　在窗口中排序数据

在 OVER 子句中可能放入的东西并非只有 PARTITION BY 子句。有时候有必要在一个窗口中排序数据。ORDER BY 将以一种特定的方式为聚集函数提供数据，例如：

```
test=# SELECT country, year, production,
   min(production) OVER (PARTITION BY country ORDER BY year)
FROM      t_oil
WHERE     year BETWEEN 1978 AND 1983
          AND country IN ('Iran', 'Oman');
```

```
 country | year | production | min
---------+------+------------+------
 Iran    | 1978 |       5302 | 5302
 Iran    | 1979 |       3218 | 3218
 Iran    | 1980 |       1479 | 1479
 Iran    | 1981 |       1321 | 1321
 Iran    | 1982 |       2397 | 1321
 Iran    | 1983 |       2454 | 1321
 Oman    | 1978 |        314 | 314
 Oman    | 1979 |        295 | 295
 Oman    | 1980 |        285 | 285
 Oman    | 1981 |        330 | 285
...
```

对于 1978 年到 1983 年这个时间段，从我们的数据集中选中了两个国家（伊朗和阿曼）。记住，在 1979 年伊朗发生一次革命，所以对其原油产量有一些影响，上述数据反映了这一情况。

这个查询所做的事情是计算到时间序列中一个特定时间点为止的最小产量。“到目前为止”是让 SQL 初学者记住 OVER 子句中 ORDER BY 子句行为的最好方式。在这个例子中，PARTITION BY 子句将为每个国家创建一个分组并且在分组中排序数据。min 函数将在有序的数据上循环并且提供所需的最小值。

如果读者之前对窗口函数并不熟悉，那么应该注意的是，使用 ORDER BY 子句所产生的效果确实不同：

```
test=# SELECT country, year, production,
          min(production) OVER (),
          min(production) OVER (ORDER BY year)
   FROM   t_oil
   WHERE  year BETWEEN 1978 AND 1983
          AND country = 'Iran';
 country | year | production | min  | min
---------+------+------------+------+------
 Iran    | 1978 |       5302 | 1321 | 5302
 Iran    | 1979 |       3218 | 1321 | 3218
 Iran    | 1980 |       1479 | 1321 | 1479
 Iran    | 1981 |       1321 | 1321 | 1321
 Iran    | 1982 |       2397 | 1321 | 1321
 Iran    | 1983 |       2454 | 1321 | 1321
(6 rows)
```

如果使用的聚集不带 ORDER BY，它将自动地取窗口中整个数据集的最小值。而有

ORDER BY 时就不是这样：它将总是给定顺序中到目前为止的最小值。

4.4.3　使用滑动窗口

到目前为止，我们所使用的窗口都是静态的。然而，对于移动平均这类计算来说这还不够。移动聚集需要一个滑动窗口，滑动窗口会随着数据的处理而移动。

这里有一个如何实现移动平均的例子：

```
test=# SELECT country, year, production,
   min(production) OVER (PARTITION BY country
                         ORDER BY year ROWS
                         BETWEEN 1 PRECEDING
                         AND 1 FOLLOWING)
   FROM t oil
   WHERE year BETWEEN 1978 AND 1983
         AND country IN ('Iran', 'Oman');
 country | year | production | min
---------+------+------------+------
 Iran    | 1978 |       5302 | 3218
 Iran    | 1979 |       3218 | 1479
 Iran    | 1980 |       1479 | 1321
 Iran    | 1981 |       1321 | 1321
 Iran    | 1982 |       2397 | 1321
 Iran    | 1983 |       2454 | 2397
 Oman    | 1978 |        314 | 295
 Oman    | 1979 |        295 | 285
 Oman    | 1980 |        285 | 285
 Oman    | 1981 |        330 | 285
 Oman    | 1982 |        338 | 330
 Oman    | 1983 |        391 | 338
(12 rows)
```

最重要的事情是移动窗口应该与 ORDER BY 子句一起使用，否则就会有大量问题。PostgreSQL 实际上会接受那样的查询，但是其结果会是彻头彻尾的垃圾。记住，将事先没有排序的数据交给一个滑动窗口只会导致随机数据。

“ROWS BETWEEN 1 PRECEDING and 1 FOLLOWING 1” 定义了窗口。在笔者的例子中，被使用的最多有 3 行：当前行、当前行的前一行以及当前行的后一行。为了展示滑动窗口如何工作，试试下面的例子：

```
test=# SELECT *,
          array_agg(id) OVER
```

```
                  (ORDER BY id ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
   FROM generate_series(1, 5) AS id;
 id | array_agg
----+-----------
 1  | {1,2}
 2  | {1,2,3}
 3  | {2,3,4}
 4  | {3,4,5}
 5  | {4,5}
(5 rows)
```

array_agg 函数将把一个值列表转变成一个 PostgreSQL 数组，它将有助于解释滑动窗口如何操作。

实际上这个微不足道的查询却有一些非常重要的方面。读者所看到的是第一个数组只包含两个值。在 1 之前没有项，因此该数组并未充满。PostgreSQL 并未增加空值项，因为它们会被聚集忽略。同样的事情发生在数据的末尾。

不过，滑动窗口还提供了更多东西。有一些关键词可以被用来指定滑动窗口：

```
test=# SELECT *,
          array_agg(id) OVER
                (ORDER BY id ROWS BETWEEN UNBOUNDED PRECEDING
                 AND 0 FOLLOWING)
   FROM generate_series(1, 5) AS id;
 id |  array_agg
----+-------------
 1  | {1}
 2  | {1,2}
 3  | {1,2,3}
 4  | {1,2,3,4}
 5  | {1,2,3,4,5}
(5 rows)
```

UNBOUNDED PRECEDING 指定当前行之前的所有东西都将在该窗口中。与 UNBOUNDED PRECEDING 相对的是 UNBOUNDED FOLLOWING：

```
test=# SELECT *,
          array_agg(id) OVER
                 (ORDER BY id ROWS BETWEEN 2 FOLLOWING
                  AND UNBOUNDED FOLLOWING)
   FROM generate_series(1, 5) AS id;
```

```
 id | array_agg
----+-----------
 1  | {3,4,5}
 2  | {4,5}
 3  | {5}
 4  |
 5  |
(5 rows)
```

如读者所见，也可以使用一个表示未来的窗口，PostgreSQL 在这一点上非常灵活。

4.4.4　提取窗口子句

窗口函数允许我们在结果集中增加联机计算的列。然而，经常发生的事情是很多这样的列都基于同一个窗口。把同样的子句一遍又一遍地写在查询中绝对不是什么好主意，因为这样会让查询变得很难阅读和维护。

WINDOW 子句允许开发者预定义一个窗口，并且将它用在查询中的多个位置。例如：

```
SELECT country, year, production, min(production) OVER (w), max(production)
OVER (w)
FROM      t_oil
WHERE     country = 'Canada'
          AND year BETWEEN 1980 AND 1985
WINDOW w AS (ORDER BY year);
 country | year | production |  min | max
---------+------+------------+------+------
 Canada  | 1980 |       1764 | 1764 | 1764
 Canada  | 1981 |       1610 | 1610 | 1764
 Canada  | 1982 |       1590 | 1590 | 1764
 Canada  | 1983 |       1661 | 1590 | 1764
 Canada  | 1984 |       1775 | 1590 | 1775
 Canada  | 1985 |       1812 | 1590 | 1812
(6 rows)
```

这个例子显示 min 和 max 将使用同一个子句。

当然，可以有多于一个 WINDOW 子句——PostgreSQL 在这里未对用户施以严格的限制。

4.4.5　使用内建窗口函数

在为读者介绍了基本概念之后，现在就可以看看 PostgreSQL 支持哪些可以立即使用

的窗口函数。读者已经见过了使用所有标准聚集函数的窗口，在那些函数之上，PostgreSQL 还提供了一些附加函数，它们只用于窗口和分析。

在本节中将解释并讨论一些非常重要的函数。

1. rank 和 dense_rank 函数

在笔者而言，rank 和 dense_rank 函数是最为突出的函数。rank 函数返回当前行在其窗口中的编号，从 1 开始计数。

这里有一个例子：

```
test=# SELECT year, production,
          rank() OVER (ORDER BY production)
   FROM t oil
   WHERE country = 'Other Middle East'
   ORDER BY rank
   LIMIT 7;
 year | production | rank
------+------------+------
 2001 |         47 |    1
 2004 |         48 |    2
 2002 |         48 |    2
 1999 |         48 |    2
 2000 |         48 |    2
 2003 |         48 |    2
 1998 |         49 |    7
(7 rows)
```

rank 列将会对那些行编号。注意例子中很多行是相等的。因此 rank 将从 2 直接跳到 7。如果想避免这样的事情，可以使用 dense_rank 函数：

```
test=# SELECT year, production,
          dense_rank() OVER (ORDER BY production)
   FROM t_oil
   WHERE country = 'Other Middle East'
   ORDER BY dense rank
   LIMIT 7;
 year | production | dense_rank
------+------------+------------
 2001 |         47 |          1
 2004 |         48 |          2
...
```

```
 2003 |          48 |          2
 1998 |          49 |          3
(7 rows)
```

PostgreSQL 会把编号弄得更加紧密，中间不会有间隔。

2. ntile 函数

一些应用要求把数据划分成完美相等的分组，ntile 函数可以做到这一点。

下面的例子展示了数据如何被划分成分组：

```
test=# SELECT year, production,
         ntile(4) OVER (ORDER BY production)
   FROM   t oil
   WHERE  country = 'Iraq'
          AND year BETWEEN 2000 AND 2006;
 year | production | ntile
------+------------+-------
 2003 |       1344 |     1
 2005 |       1833 |     1
 2006 |       1999 |     2
 2004 |       2030 |     2
 2002 |       2116 |     3
 2001 |       2522 |     3
 2000 |       2613 |     4
(7 rows)
```

该查询把数据划分成 4 个分组。问题在于这里只选择了 7 行，这样就不可能创建 4 个平均的分组。如你所见，PostgreSQL 将填满前三个分组并且让最后一个分组小一点。可以确信末尾的分组将总是比其他分组小一点。



这个例子中只用了少量的行。在实际应用中将涉及数百万行，因此如果分组不完美地相等也没有问题。

ntile 函数通常不会被单独使用。当然，它有助于为一行分配一个分组 ID。不过，在实际应用中，人们想要在那些分组之上执行计算。假定用户想要为其数据创建一种分位数分布，下面的例子可以做到：

```
test=# SELECT grp, min(production), max(production), count(*)
FROM (
SELECT year, production,
         ntile(4) OVER (ORDER BY production) AS grp
```

```
FROM t_oil
WHERE country = 'Iraq'
) AS x
GROUP BY ROLLUP (1);
 grp | min   | max   | count
-----+-------+------+-------
 1   | 285   | 1228 |    12
 2   | 1313 | 1977 |    12
 3   | 1999 | 2422 |    11
 4   | 2428 | 3489 |    11
     | 285   | 3489 |    46
(5 rows)
```

最重要的是该计算不能在一步中完成。当笔者在 Cybertec（www.cybertec.at）上做 SQL 培训课程时，笔者尝试向学员解释，只要不知道如何一次完成全部工作，那就考虑使用一个子查询。在分析工作中这通常是一个好主意。在这个例子中，所做的第一件事（在子查询中）是为每一组贴一个分组标签，然后在主查询中取得那些分组并且进行处理。

最终的结果就已经是可以在实际应用（可能是图表旁边的图例）中使用的东西。

3. lead 和 lag 函数

虽然 ntile 函数对于把一个数据集划分为分组很重要，这里的 lead 和 lag 函数可以在结果集内移动行。一种典型的用例是计算一年和接下来一年的产量差：

```
test=# SELECT year, production,
          lag(production, 1) OVER (ORDER BY year)
   FROM t_oil
   WHERE country = 'Mexico'
   LIMIT 5;
 year | production | lag
------+------------+-----
 1965 |        362 |
 1966 |        370 | 362
 1967 |        411 | 370
 1968 |        439 | 411
 1969 |        461 | 439
(5 rows)
```

在实际计算产量中的变化之前，还是先看看 lag 函数会做什么。读者可以看到列被按照一行移动。数据按照 ORDER BY 子句的定义进行移动。在笔者的例子中，它表示向下。ORDER BY DESC 子句当然会让数据向上移动。

有了这个基础后，查询就很容易了：

```
test=# SELECT year, production,
          production - lag(production, 1) OVER (ORDER BY year)
   FROM   t_oil
   WHERE  country = 'Mexico'
   LIMIT 3;
 year | production | ?column?
------+------------+----------
 1965 |        362 |
 1966 |        370 |        8
 1967 |        411 |       41
(3 rows)
```

所有要做的就是计算差异而已，就好像有了另一列一样。注意，lag 函数有两个参数。第一个参数表示要显示哪一列，第二个参数告诉 PostgreSQL 想要移动多少行。因此放上 7 就表示对任何行都要减去 7 个行。



第一个值为空（其他没有前序值的延迟行也是如此）。

lead 函数是 lag 函数的反面：它将把行向上移而不是向下移：

```
test=# SELECT year, production,
          production - lead(production, 1) OVER (ORDER BY year)
FROM      t oil
WHERE     country = 'Mexico'
LIMIT 3;
 year | production | ?column?
------+------------+----------
 1965 |        362 |       -8
 1966 |        370 |      -41
 1967 |        411 |      -28
(3 rows)
```

基本上，对于 lead 和 lag 列，PostgreSQL 也会接受负值。因此，lag(production, -1)是 lead(production,1)的一种替代品。不过，使用正确的函数把数据移向用户想要的方向会更加清晰。

到目前为止，读者已经看到了如何延迟单个列。在大部分应用中延迟单个值将是大部分开发人员使用的标准情况。关键是，PostgreSQL 还能做更多事情。例如可以延迟整个行：

```
test=# \x
Expanded display is on.
test=# SELECT year, production,
          lag(t oil, 1) OVER (ORDER BY year)
   FROM   t oil
   WHERE country = 'USA'
   LIMIT 3;
-[ RECORD 1 ]-------------------------------------
year       | 1965
production | 9014
lag        |
-[ RECORD 2 ]-------------------------------------
year       | 1966
production | 9579
lag        | ("North America",USA,1965,9014,11522)
-[ RECORD 3 ]-------------------------------------
year       | 1967
production | 10219
lag        | ("North America",USA,1966,9579,12100)
```

这里最美妙的事情是，可以把不止一个单值拿来与以前的行比较。但问题是 PostgreSQL 只会把整个行作为一种组合类型返回，因此很难去处理它。为了分解一种组合类型，可以使用圆括号和一个星号：

```
test=# SELECT year, production,
          (lag(t oil, 1) OVER (ORDER BY year)).*
   FROM t oil
   WHERE country = 'USA'
   LIMIT 3;
 year | prod  |   region    | country | year | prod | consumption
------+-------+-------------+---------+------+------+-------------
 1965 | 9014  |             |         |      |      |
 1966 | 9579  | N. America | USA     | 1965 | 9014 |       11522
 1967 | 10219 | N. America | USA     | 1966 | 9579 |       12100
(3 rows)
```

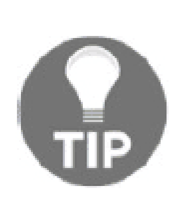
这有什么用呢？延迟一整个行将会让它无法查看数据是否被插入多次。它对于在时间序列中检测重复行（或者接近于重复的行）来说非常简单。

看看下面的例子：

```
test=# SELECT * FROM (
SELECT t_oil, lag(t_oil)
```

```
OVER (ORDER BY year)
FROM t_oil
WHERE country = 'USA'
) AS x
WHERE t oil = lag;
 t oil | lag
-------+-----
(0 rows)
```

当然，样例数据中并不包含重复。但是在实际的例子中，重复很容易发生，并且即便没有主键也很容易检测到它们。



t_oil 实际上是一整行。子查询返回的 lag 也是一个完整的行。在 PostgreSQL 中，在组成域相同的情况下可以直接比较组合类型。PostgreSQL 将会逐个比较其中的域。

4. first_value、nth_value 和 last_value 函数

有时，有必要基于一个数据窗口的第一个值来计算数据。做这件事情的函数无疑就是 first_value：

```
test=# SELECT year, production,
                 first value(production) OVER (ORDER BY year)
   FROM t oil
   WHERE country = 'Canada'
   LIMIT 4;
 year | production | first value
------+------------+-------------
 1965 |        920 |         920
 1966 |       1012 |         920
 1967 |       1106 |         920
 1968 |       1194 |         920
(4 rows)
```

同样，需要一个排序顺序来告诉系统第一个值究竟在哪里。然后 PostgreSQL 将把同样的值放在最后一列中。如果想要找到窗口中的最后一个值，只需要用 last_value 函数代替 first_value。

如果用户的兴趣不在于第一个或者最后一个值，而是想要找某个在中间的值，PostgreSQL 还提供了 nth_value 函数：

```
test=# SELECT year, production,
          nth_value(production, 3) OVER (ORDER BY year)
   FROM t oil
   WHERE country = 'Canada';
 year | production | nth value
------+------------+-----------
 1965 |        920 |
 1966 |       1012 |
 1967 |       1106 |      1106
 1968 |       1194 |      1106
...
```

在这个例子中，第三个值将被放到最后一列中，不过，注意前两行的最后一列是空的。问题在于当 PostgreSQL 开始处理数据时，第三个值还不知道。因此，对前两行就会放入空值。现在的问题是：我们怎样才能让时间序列更加完整，并且把那两个空值替换成后面要到来的数据？

这里有一种方法：

```
test=# SELECT *,
          min(nth_value) OVER ()
FROM (
SELECT year, production,
nth value(production, 3) OVER (ORDER BY year)
FROM t oil
WHERE country = 'Canada'
) AS x
LIMIT 4;
 year | production | nth_value | min
------+------------+-----------+------
 1965 |        920 |           | 1106
 1966 |       1012 |           | 1106
 1967 |       1106 |      1106 | 1106
 1968 |       1194 |      1106 | 1106
(4 rows)
```

子查询将创建不完整的时间序列，顶层的 SELECT 将补全数据。这里的情节是：只是补全数据可能会更复杂，因此比起一步做完来说，一个子查询可能会带来一些机会，增加某种更复杂的逻辑。

5. row_number 函数

本节中要讨论的最后一个函数是 row_number，它可以被用来返回一个虚拟 ID。听起

来简单吧，下面是它的例子：

```
test=# SELECT country, production,
              row_number() OVER (ORDER BY production)
    FROM t_oil
    LIMIT 3;
 country | production | row number
---------+------------+------------
 Yemen   |         10 |          1
 Syria   |         21 |          2
 Yemen   |         26 |          3
(3 rows)
```

row_number 函数简单地为行分配一个编号，其中绝对不会有重复。

这里有趣的一点是即使没有排序（如果顺序没有关系）它也能工作：

```
test=# SELECT country, production,
              row number() OVER ()
    FROM t oil
    LIMIT 3;
 country | production | row number
---------+------------+------------
 USA     |       9014 |          1
 USA     |       9579 |          2
 USA     |      10219 |          3
(3 rows)
```

4.5　编写自己的聚集

在本书中用到的大部分都是 PostgreSQL 系统提供的函数。不过，SQL 提供的东西可能并不能完全满足用户。好消息是可以向数据库引擎增加自己的聚集。在本节中读者将会学到如何做这件事。

4.5.1　创建简单的聚集

这个例子的目标是解决一个非常简单的问题：如果叫了一辆出租车，通常必须支付起步价（例如，2.50 欧元）。那么让我们假设每公里顾客需要支付 2.20 欧元。现在问题来了：一趟行程的总价是多少？

当然，这个例子简单到完全可以不用自定义聚集来解决，不过，还是让我们看看用

自定义聚集应该怎么做。首先创建一些测试数据：

```
test=# CREATE TABLE t_taxi (trip_id int, km numeric);
CREATE TABLE
test=# INSERT INTO t_taxi VALUES
   (1, 4.0), (1, 3.2), (1, 4.5),
   (2, 1.9), (2, 4.5);
INSERT 0 5
```

PostgreSQL 提供了 CREATE AGGREGATE 命令来创建聚集。这个命令的语法已经变得如此强大和冗长，因此在本书中包括\h 的输出已经没有任何意义。笔者推荐去参考 PostgreSQL 文档（可以在 https://www.postgresql.org/docs/devel/static/sql-createaggregate.html 找到）。

编写一个聚集所需要的第一个部分是一个函数，每一行都会调用它。该函数将接受一个中间值和取自被处理行的数据。这里是一个例子：

```
test=# CREATE FUNCTION taxi_per_line (numeric, numeric)
RETURNS numeric AS
$$
BEGIN
RAISE NOTICE 'intermediate: %, per row: %', $1, $2;
RETURN $1 + $2*2.2;
END;
$$ LANGUAGE 'plpgsql';
CREATE FUNCTION
```

现在已经可以创建一个简单的聚集：

```
test=# CREATE AGGREGATE taxi_price (numeric)
(
          INITCOND = 2.5,
          SFUNC = taxi per line,
          STYPE = numeric
);
CREATE AGGREGATE
```

如前所述，每一次行程从踏进出租车开始价格就是 2.50 欧元，这由 INITCOND（初始条件）定义，它表示每个分组的起始值。然后对分组中的每一行都调用一个函数，在笔者的例子中这个函数是已经定义好的 taxi_per_line。如你所见，它需要两个参数。第一个参数是一个中间值。那些额外的参数（可能有很多）是用户传递给函数的。

下面的语句展示了什么时间以及怎样传入哪些数据：

```
test=# SELECT trip_id, taxi_price(km)
          FROM t_taxi
          GROUP BY 1;
NOTICE: intermediate: 2.5, per row: 4.0
NOTICE: intermediate: 11.30, per row: 3.2
NOTICE: intermediate: 18.34, per row: 4.5
NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
 trip_id | taxi_price
---------+------------
 1       |      28.24
 2       |      16.58
(2 rows)
```

系统从行程一和 2.50 欧元（初始条件）开始。然后增加了 4 公里。现在的总价是 2.50 + 4×2.2。然后加入下一行，这会增加 3.2×2.2，以此类推。因此第一趟行程费用是 28.24。

然后下一趟行程开始。同样，它开始于一个新的初始条件并且 PostgreSQL 会为每一行调用一次函数。

在 PostgreSQL 中一个聚集也可以自动地作为窗口函数使用，不需要任何额外的步骤——用户可以直接使用该聚集：

```
test=# SELECT *, taxi_price(km) OVER (PARTITION BY trip_id ORDER BY km)
FROM  t_taxi;
NOTICE: intermediate: 2.5, per row: 3.2
NOTICE: intermediate: 9.54, per row: 4.0
NOTICE: intermediate: 18.34, per row: 4.5
NOTICE: intermediate: 2.5, per row: 1.9
NOTICE: intermediate: 6.68, per row: 4.5
 trip_id |  km  | taxi_price
---------+-----+------------
       1 | 3.2 |       9.54
       1 | 4.0 |      18.34
       1 | 4.5 |      28.24
       2 | 1.9 |       6.68
       2 | 4.5 |      16.58
(5 rows)
```

这个查询给出的是到行程中一个给定点时的价格是多少。

我们已经定义的聚集将在每一行上调用一次函数。不过，用户怎样才能计算平均值或者类似的值？只有增加一个 FINALFUNC 计算才能做到。为了展示 FINALFUNC，可

以扩展一下刚才的例子。假定想要在下车时给司机 10%的小费，这 10%必须在总价钱知道之后立刻加到总价上，这就是 FINALFUNC 介入的时机。下面将展示其实现：

```
test=# DROP AGGREGATE taxi_price(numeric);
DROP AGGREGATE
```

首先删除旧的聚集，然后定义 FINALFUNC，它会得到中间结果作为参数然后玩些魔术：

```
test=# CREATE FUNCTION taxi_final (numeric)
   RETURNS numeric AS
$$
   SELECT $1 * 1.1;
$$
LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

在这个例子中计算非常简单——将 10%加到总和上。

一旦该函数被部署好，就可以重新创建聚集：

```
test=# CREATE AGGREGATE taxi_price (numeric)
   (
         INITCOND = 2.5,
         SFUNC = taxi_per_line,
         STYPE = numeric,
         FINALFUNC = taxi_final
);
CREATE AGGREGATE
```

最后，得到的价格将比以前的要高一点：

```
test=# SELECT trip_id, taxi_price(km)
   FROM t_taxi
   GROUP BY 1;
NOTICE: intermediate: 2.5, per row: 4.0
...
 trip_id | taxi_price
---------+------------
       1 |     31.064
       2 |     18.238
(2 rows)
```

PostgreSQL 会自动搞定所有的分组之类的工作。

对于简单的计算，中间结果可以使用简单数据类型。但是，不是所有的操作都可以通过只传递简单数字和文本就能解决。幸运的是，PostgreSQL 允许使用组合数据类型，它也能用作中间结果。

想象一下用户想要计算某种数据（可能是一种时间序列或者其他）的均值。中间结果可能长这样：

```
test=# CREATE TYPE my intermediate AS (c int4, s numeric);
CREATE TYPE
```

请放心地组合任意类型来服务于你的程序，只需要把它作为第一参数传递给聚集函数，并且按照需要把数据作为额外的参数加入即可。

4.5.2　为并行查询增加支持

读者已经看到的只是一个简单的聚集，它不能支持并行查询。为了解决这一挑战，接下来的几个例子全都与改进和提速有关。

在创建聚集时，用户可以有选择地定义下列选项：

```
[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
```

默认情况下聚集不支持并行查询。不过，出于性能原因，明确地说明聚集能干什么是有意义的。

- UNSAFE：这种模式下不允许并行查询。
- RESTRICTED：这种模式下聚集可以在并行模式中执行，但是只限于在并行分组领导者中执行。
- SAFE：提供了对并行查询的完全支持。

如果把一个函数标记为 SAFE，必须记住该函数不能有副作用，执行顺序不能对查询的结果造成影响。只有这样，PostgreSQL 才应该被允许并行执行操作。无副作用的函数的例子有 sin(x)、length(s)等。IMMUTABLE 函数是很好的候选，因为它们被保证对给定的相同输入返回相同的结果。如果应用某种限制，STABLE 函数可以并行使用。

4.5.3　改进效率

到目前为止，所定义的聚集功能已经很强大。但是，如果使用滑动窗口，函数调用的数量就会爆炸性地增长。从下面这个例子可以看出：

```
test=# SELECT taxi_price(x::numeric)
          OVER ( ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
   FROM generate_series(1, 5) AS x;
```

```
NOTICE: intermediate: 2.5, per row: 1
NOTICE: intermediate: 4.7, per row: 2
NOTICE: intermediate: 9.1, per row: 3
NOTICE: intermediate: 15.7, per row: 4
NOTICE: intermediate: 2.5, per row: 2
NOTICE: intermediate: 6.9, per row: 3
NOTICE: intermediate: 13.5, per row: 4
NOTICE: intermediate: 22.3, per row: 5
...
```

对于每一行，PostgreSQL 都将处理整个窗口。如果滑动窗口很大，效率将会每况愈下。为了解决这一问题，可以扩展我们的聚集。在此之前还是要删除掉旧的聚集：

```
DROP AGGREGATE taxi_price(numeric);
```

基本上需要两个函数：msfunc 函数将把窗口中的下一行加入中间结果上：

```
CREATE FUNCTION taxi_msfunc(numeric, numeric)
RETURNS numeric AS
$$
        BEGIN
          RAISE NOTICE 'taxi_msfunc called with % and %', $1, $2;
          RETURN $1 + $2;
        END;
$$ LANGUAGE 'plpgsql' STRICT;
```

minvfunc 函数将从中间结果中去掉刚移出窗口之外的值：

```
CREATE FUNCTION taxi_minvfunc(numeric, numeric)
RETURNS numeric AS
$$
        BEGIN
          RAISE NOTICE 'taxi_minvfunc called with % and %', $1, $2;
          RETURN $1 - $2;
        END;
$$ LANGUAGE 'plpgsql' STRICT;
```

在笔者的例子中用的是加和减。在更加精致的例子中，这种计算可能是任意的复杂度。

下一个语句展示了如何重建聚集：

```
CREATE AGGREGATE taxi_price (numeric)
(
        INITCOND = 0,
```

```
        STYPE = numeric,
        SFUNC = taxi_per_line,
        MSFUNC = taxi_msfunc,
        MINVFUNC = taxi_minvfunc,
        MSTYPE = numeric
);
```

现在让我们再次运行同一个查询：

```
test# SELECT taxi_price(x::numeric)
        OVER (ROWS BETWEEN 0 FOLLOWING AND 3 FOLLOWING)
FROM    generate_series(1, 5) AS x;
NOTICE: taxi_msfunc called with 1 and 2
NOTICE: taxi_msfunc called with 3 and 3
NOTICE: taxi_msfunc called with 6 and 4
NOTICE: taxi_minfunc called with 10 and 1
NOTICE: taxi_msfunc called with 9 and 5
NOTICE: taxi_minfunc called with 14 and 2
NOTICE: taxi_minfunc called with 12 and 3
NOTICE: taxi_minfunc called with 9 and 4
```

函数调用的数量已经急剧下降。对每一行只有固定的几次调用需要被执行，这种情况下已经不再需要一次又一次地重复计算同一个帧了。

4.5.4　编写假想聚集

编写聚集并不难，并且它对于执行更加复杂的操作很有好处。本节的计划是编写一个假想聚集，其概念已经在本章中讨论过。

实现假想聚集与编写普通聚集没什么太多不同，真正难的部分是找出何时真正需要用到它。为了尽可能让本节容易理解，笔者决定包括一个小例子：给定一个特定的顺序，如果我们把 abc 添加到字符串的尾部结果会是什么？

下面是实现：

```
CREATE AGGREGATE name ([[argmode] [argname] arg_data_type [ , ... ] ]
                        ORDER BY [ argmode ] [ argname ] arg_data_type
[ , ... ] ) (
   SFUNC = sfunc,
   STYPE = state_data_type
   [ , SSPACE = state_data_size ]
   [ , FINALFUNC = ffunc ]
   [ , FINALFUNC_EXTRA ]
```

```
   [ , INITCOND = initial_condition ]
   [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
   [ , HYPOTHETICAL ]
)
```

需要两个函数。sfunc 函数会为每一行调用：

```
CREATE FUNCTION hypo_sfunc(text, text)
RETURNS text AS
$$
BEGIN
     RAISE NOTICE 'hypo_sfunc called with % and %', $1, $2;
     RETURN $1 || $2;
   END;
$$ LANGUAGE 'plpgsql';
```

两个参数将被传递给这个过程，其逻辑和之前相同。和先前类似，可以定义一个 final 函数调用：

```
CREATE FUNCTION hypo_final(text, text, text)
    RETURNS text AS
$$
BEGIN
     RAISE NOTICE 'hypo_final called with %, %, and %',
          $1, $2, $3;
     RETURN $1 || $2;
   END;
$$ LANGUAGE 'plpgsql';
```

一旦这些函数就位就可以创建假想聚集：

```
CREATE AGGREGATE whatif(text ORDER BY text) (
   INITCOND = 'START',
   STYPE = text,
   SFUNC = hypo_sfunc,
   FINALFUNC = hypo_final,
   FINALFUNC_EXTRA = true,
   HYPOTHETICAL
);
```

注意，这个聚集已经被标记为 HYPOTHETICAL，因此 PostgreSQL 将会知道它实际是一种什么样的聚集。

创建好聚集后，就可以运行它：

```
test=# SELECT whatif('abc'::text) WITHIN GROUP (ORDER BY id::text)
   FROM generate_series(1, 3) AS id;
NOTICE: hypo sfunc called with START and 1
NOTICE: hypo sfunc called with START1 and 2
NOTICE: hypo sfunc called with START12 and 3
NOTICE: hypo final called with START123, abc, and <NULL>
  whatif
-------------
 START123abc
(1 row)
```

理解所有那些聚集的关键实际上在于充分地观察何时何种类型的函数会被调用，以及总体机制如何工作。

4.6　总　结

在本章中，读者学到了 SQL 提供的高级特性。在简单聚集之上 PostgreSQL 提供了有序集、分组集、窗口函数、递归以及创建自定义聚集的接口。在数据库中运行聚集的好处是代码容易编写并且数据库引擎通常有效率优势。

在第 5 章中，我们将把注意力转向更多管理任务上，例如处理日志、理解系统统计信息以及实现监控。

第 5 章　日志文件和系统统计信息

在第 4 章，读者从一个不同的角度学到了很多有关高级 SQL 的知识和理解 SQL 的方法。不过，数据库工作并不只是设计奇特的 SQL，有时候数据库工作也涉及保持数据库以专业的方式运行。要做到这一点，非常重要的是密切关注系统统计信息、日志文件等等。监控是专业地运行数据库的关键。

在本章，读者将会学到以下主题：

- 收集运行时统计信息。
- 创建日志文件。
- 收集重要信息。
- 理解数据库统计信息。

5.1　收集运行时统计信息

读者一定要学会的第一点是使用和理解 PostgreSQL 已经提供的内建统计信息。正如笔者一直在讲的，如果不首先收集数据来做出精明的决定，就没有办法改进性能和可靠性。

本节将指引读者了解 PostgreSQL 的运行时统计信息并且详细解释如何从数据库设置中提取更多数据。

- 使用 PostgreSQL 系统视图

PostgreSQL 提供了一大堆系统视图，它们让管理员和开发者之类的人可以深入地审视系统中究竟在做什么。其问题在于，很多人实际上收集了所有这类数据，但却无法真正理解它。通常来说：对无法理解的东西绘制图表是毫无意义的。因此，本节的目标是讲清楚 PostgreSQL 必须提供哪些东西，以期能让人们充分利用它们为自己的工作服务。

1. 检查实时流量

每当笔者检查一个系统时，在进一步深究之前，笔者总是首先会检查一个系统视图。当然，笔者说的就是 pg_stat_activity。该视图的思想就是让用户有机会弄明白现在正在发生些什么。

下面是该视图的结构：

```
test=# \d pg_stat_activity
          View "pg_catalog.pg_stat_activity"
      Column      |           Type           | Modifiers
------------------+--------------------------+-----------
 datid            | oid                      |
 datname          | name                     |
 pid              | integer                  |
 usesysid         | oid                      |
 usename          | name                     |
 application_name | text                     |
 client_addr      | inet                     |
 client hostname  | text                     |
 client port      | integer                  |
 backend start    | timestamp with time zone |
 xact start       | timestamp with time zone |
 query_start      | timestamp with time zone |
 state_change     | timestamp with time zone |
 wait_event_type  | text                     |
 wait event       | text                     |
 state            | text                     |
 backend xid      | xid                      |
 backend xmin     | xid                      |
 query            | text                     |
```

在 pg_stat_activity 将为用户提供的数据中，每个活动连接都有一行。用户将会看到数据库的内部对象 ID（datid）、被某人连接的数据库名称以及服务于这个连接的进程 ID（pid）。在此之上，PostgreSQL 还将告诉用户谁在连接（usename，注意其中少了一个 r）以及该用户的内部对象 ID（usesysid）。

其中还有一个名为 application_name 的域值得多说几句。通常，application_name 可以由最终用户自由地设置：

```
test=# SET application name TO 'postgresql-support.de';
SET
test=# SHOW application name ;
   application name
-----------------------
 postgresql-support.de
(1 row)
```

其关键在于，假定有数千个连接来自于一个 IP，作为一个管理员是否能知道一个特定连接现在实际在做什么？管理员不可能凭记忆知道所有的 SQL。如果客户端能善意地

设置 application_name 参数，就能更容易地理解一个连接的真正目的。在笔者的例子中，笔者将这一名称设置在连接所属的域。这使得查找相似的连接更容易，因为相似的操作也会导致类似的问题。

接下来的 3 个列（client_）将告诉用户一个连接来自何方。PostgreSQL 将显示 IP 地址甚至（如果被配置）主机名。

backend_start 将会告诉用户一个特定连接何时被启动。xact_start 表示一个事务何时被启动。接着还有 query_start 和 state_change。在“黑暗”的旧时代，PostgreSQL 只会显示活动查询。在那个时期查询运行时间比现如今更长，这当然说得通。在现代硬件上，OLTP 查询可能只会消耗不到一毫秒，因此很难捕捉到可能做了坏事的查询。其解决方案是既显示活动查询又显示正查看的连接之前执行的查询。

用户可能会看到这样的数据：

```
test=# SELECT pid, query_start, state_change, state, query FROM
pg stat activity;
...
-[ RECORD 2 ]+--------------------------------------------------------
--
pid          | 28001
query_start  | 2016-11-05 10:03:57.575593+01
state_change | 2016-11-05 10:03:57.575595+01
state        | active
query        | SELECT pg_sleep(10000000);
```

在这种情况下，用户可以看到在第二个连接中正在执行 pg_sleep。

一旦这个查询被中止，输出将会改变：

```
-[ RECORD 2 ]+--------------------------------------------------------
--
pid          | 28001
query start  | 2016-11-05 10:03:57.575593+01
state_change | 2016-11-05 10:05:10.388522+01
state        | idle
query        | SELECT pg_sleep(10000000);
```

该查询现在被标记为 idle。state_change 和 query_start 之间的差是该查询执行所需的时间。

因此，pg_stat_activity 让用户对当前系统正在做什么有一个大体上的认识。新的 state_change 域让我们更有可能找出开销巨大的查询。

现在的问题是，一旦找到了不好的查询，如何能真正除去它们？PostgreSQL 提供了两

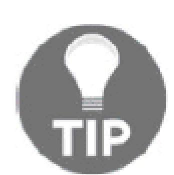
个函数来处理这些事情：pg_cancel_backend 和 pg_terminate_backend。pg_cancel_backend 函数将中止查询但是把连接留在原地。pg_terminate_backend 函数更加激进一些，它将把查询和整个数据库连接全都清除。

如果用户想要断开除自己之外所有其他用户的连接，可以这样做：

```
test=# SELECT pg terminate backend(pid) FROM pg stat activity WHERE pid <>
pg backend pid();
 pg terminate backend
----------------------
 t
 t
(2 row)
```

如果用户正好被踢出了系统，将会显示下列消息：

```
test=# SELECT pg_sleep(10000000);
FATAL: terminating connection due to administrator command
server closed the connection unexpectedly
   This probably means the server terminated abnormally
   before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```



只有 psql 会尝试重新连接。对于大部分其他客户端来说不是这样，特别是客户端库不会尝试重新连接。

2. 检查数据库

在检查了活动数据库连接之后，就可以开始更进一步检查数据库层面的统计信息。pg_stat_database 将对 PostgreSQL 实例中的每个数据库返回一行。

在其中可以看到这些：

```
test=# \d pg stat database
          View "pg_catalog.pg_stat_database"
     Column      |          Type           | Modifiers
----------------+-------------------------+-----------
 datid           | oid                      |
 datname         | name                     |
 numbackends     | integer                  |
 xact commit     | bigint                   |
 xact_rollback   | bigint                   |
 blks_read       | bigint                   |
 blks_hit        | bigint                   |
```

```
tup_returned   | bigint                   |
tup_fetched    | bigint                   |
tup_inserted   | bigint                   |
tup_updated    | bigint                   |
tup_deleted    | bigint                   |
conflicts      | bigint                   |
temp_files     | bigint                   |
temp_bytes     | bigint                   |
deadlocks      | bigint                   |
blk_read_time  | double precision         |
blk_write_time | double precision         |
stats_reset    | timestamp with time zone |
```

紧接着数据库 ID 和数据库名称是一个名为 numbackends 的列，它显示当前打开的数据库连接数。

然后是 xact_commit 和 xact_rollback，它们表示应用是否在提交或回滚。blks_hit 和 blks_read 会告诉用户缓冲命中和缓冲未命中的计数。在检查这两列时，记住我们主要在讨论共享缓冲区命中和共享缓冲区未命中。在数据库层面上没有合理的方法能区分出文件系统的缓冲命中以及实际的磁盘命中。在 Cybertec（www.postgresql-support.de），我们喜欢把磁盘等待和 pg_stat_database 中的缓冲未命中关联在一起研究系统中究竟在做什么。

tup_列将告诉用户系统中是否正在进行大量的读或者大量的写。

接下来还有 temp_files 和 temp_bytes。这两列的信息惊人的重要，因为它们将告诉用户其数据库是否不得不在磁盘上写入临时文件，这将不可避免地拖慢操作。临时文件使用量大的原因可能是什么？主要原因如下。

- **过低的设置**：如果 work_mem 设置太低，就没有办法在 RAM 中做任何事情，因此 PostgreSQL 将利用磁盘。
- **愚蠢的操作**：经常会有人用相当昂贵但又毫无意义的查询来折磨他们的系统。如果在一个 OLTP 系统上看到很多临时文件，可以考虑检查一下昂贵的查询。
- **索引和其他管理任务**：有时，人们可能会创建索引或者运行 DDL。这些操作可能会导致临时文件 I/O，但（在很多情况下）不一定是问题。

简而言之，即便系统运行得很好也可能出现临时文件。但是，注意关注临时文件并且确保不会过于频繁地需要临时文件绝对是有意义的。

最后还有两个更加重要的域：blk_read_time 和 blk_write_time。默认情况下，这两个域是空的并且不会有数据被收集。它们的意图是让用户有办法能查看在 I/O 上花费了多少时间。这两个域为空的原因是参数 track_io_timing 默认被关闭。这样做是有道理的，想象一下用户想要检查读取 1000000 块需要多久。要做到这一点，需要调用 C 库中的 time 函

数两次，这样为了读取 8GB 数据就需要 2000000 次额外的函数调用。这是否将导致很多开销完全取决于用户系统的速度。

幸运的是，有一个工具能帮助用户判断计时工作有多昂贵：

```
[hs@zenbook ~]$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 23.16 nsec
Histogram of timing durations:
< usec     % of total        count
     1       97.70300    126549189
     2        2.29506      2972668
     4        0.00024          317
     8        0.00008          101
    16        0.00160         2072
    32        0.00000            5
    64        0.00000            6
   128        0.00000            4
   256        0.00000            0
   512        0.00000            0
  1024        0.00000            4
  2048        0.00000            2
```

在笔者的情况中，为一个会话或者在 postgresql.conf 文件中将 track_io_timing 打开的开销大约是 23 纳秒，这还算可以。专业的高端服务器可以提供的数字低至 14 纳秒，而虚拟化的服务器可能会返回高达 1400 纳秒甚至 1900 纳秒的值，那就真的很差了。在遇见了四位数值的情况下，度量 I/O 计时无疑会导致可察觉的开销，这将会拖慢系统。通常的规则是，在真实硬件上，计时不是问题；在虚拟系统上，在打开之前先检查一下。



也可以使用 ALTER DATABASE、ALTER USER 之类的语句有选择地打开计时。

一旦用户已经获得了其数据库中正在做什么的总体情况后，接下来的想法就是深入地查看各个表上的活动。这里有两个系统视图可以帮到我们： pg_stat_user_tables 和 pg_statio_user_tables。下面是第一个：

```
test=# \d pg_stat_user_tables
           View "pg_catalog.pg_stat_user_tables"
        Column          |            Type             | Modifiers
------------------------+-----------------------------+-----------
 relid                  | oid                         |
 schemaname             | name                        |
```

```
relname              | name                     |
seq_scan             | bigint                   |
seq_tup_read         | bigint                   |
idx_scan             | bigint                   |
idx_tup_fetch        | bigint                   |
n_tup_ins            | bigint                   |
n_tup_upd            | bigint                   |
n_tup_del            | bigint                   |
n_tup_hot_upd        | bigint                   |
n_live_tup           | bigint                   |
n_dead_tup           | bigint                   |
n_mod_since_analyze  | bigint                   |
last_vacuum          | timestamp with time zone |
last_autovacuum      | timestamp with time zone |
last_analyze         | timestamp with time zone |
last_autoanalyze     | timestamp with time zone |
vacuum_count         | bigint                   |
autovacuum_count     | bigint                   |
analyze_count        | bigint                   |
autoanalyze_count    | bigint                   |
```

在笔者看来，pg_stat_user_tables 是最重要的系统视图之一，但也是最容易被误解甚至被忽视的系统视图之一。笔者觉得很多人虽然看过它却无法充分发挥其潜力。如果使用得当，pg_stat_user_tables 在一些情况下简直是一部启示录。

在我们深入解释数据之前，重点得理解实际有哪些域。首先，对每个表都有一项，它表明发生在该表上的顺序扫描的次数（seq_scan）。然后还有 seq_tup_read，它告诉我们在那些顺序扫描时系统读取了多少个元组。



记住 seq_tup_read 列，它包含至关重要的信息，能够帮助发现性能问题。

idx_scan 是列表上的下一个，它会向我们展示多长时间为这个表使用一次索引。PostgreSQL 还将展示哪些查询返回了多少行。然后还有一些以 n_tup_开头的列，它们会告诉我们插入、更新以及删除了多少。这里最重要的事情与 HOT UPDATE 有关。在运行一个 UPDATE 时，PostgreSQL 必须复制一行以确保 ROLLBACK 能正确地工作。HOT UPDATE 的好处在于它允许 PostgreSQL 确保一行不需要离开一个块。行的备份也留在同一块中，这样通常对性能有利。适当数量的 HOT UPDATE 表示用户在 UPDATE 比较强烈的负载下走在正确的轨道上。虽然这里很难对所有的用例给出普通更新和 HOT UPDATE 之间的完美比例，但人们实际上已经开始思考，什么样的负载能从很多就地操作中获

益。通常来说，负载中 UPDATE 的强度越高，有很多 HOT UPDATE 子句就会更好。

最后还有一些 VACUUM 统计信息，其含义大部分都不言而喻。

3. 理解 pg_stat_user_tables

阅读所有这类数据可能会很有趣，但是如果无法理解它，就会发现它其实相当无聊。一种使用 pg_stat_user_tables 的方法是检测哪些表可能需要索引。一种找到正确方向线索的方法是使用下面的查询，它已经为笔者服务了多年：

```
SELECT schemaname, relname,
       seq scan,
       seq tup read,
       seq tup read / seq scan AS avg,
       idx scan
FROM   pg stat user tables
WHERE  seq_scan > 0
ORDER BY seq_tup_read
DESC LIMIT 25;
```

其思想是找出大型的表，它们被顺序扫描频繁地使用。那些表自然会出现在列表的顶部，它们会给出非常高的 seq_tup_read 值，能找到这样的表实在让人很兴奋。



从上至下观察结果并寻找昂贵的扫描。记住顺序扫描不一定是不好的，在备份、分析语句中很自然地会出现顺序扫描，因此也不会造成任何损害。不过，如果总是在运行大型的顺序扫描，系统的性能将会每况愈下。

注意这个查询的价值就像黄金一般——它将帮助用户找出缺少索引的表。近二十年的实践经验已经一次又一次地证明缺少索引是糟糕性能最重要的原因，因此读者看到的这个查询可以说就是一块金子。

一旦用户已经找到了可能缺失的索引，可以考虑简要地看看表的缓冲行为。pg_statio_user_tables 将包含关于各种东西的信息，例如表（heap_blks_）、索引（idx_blks_）以及**超尺寸属性存储技术（TOAST）**表的缓冲行为。最后用户能发现更多有关 TID 扫描的信息，但这通常与系统的总体性能无关：

```
test=# \d pg_statio_user_tables
View "pg catalog.pg statio user tables"
     Column       |  Type   | Modifiers
-----------------+---------+-----------
 relid            | oid     |
 schemaname       | name    |
```

```
 relname          | name   |
 heap_blks_read   | bigint |
 heap_blks_hit    | bigint |
 idx_blks_read    | bigint |
 idx_blks_hit     | bigint |
 toast_blks_read  | bigint |
 toast_blks_hit   | bigint |
 tidx_blks_read   | bigint |
 tidx_blks_hit    | bigint |
```

尽管 pg_statio_user_tables 包含重要的信息，但通常的情况是 pg_stat_user_tables 更可能会为用户提供真正相关的观察（例如缺失索引等）。

4. 深入研究索引

虽然 pg_stat_user_tables 对找出缺失的索引很重要，但有时有必要找出本不应存在的索引。最近，笔者因公去了德国一趟，过程中发现一个系统中包含的大部分是毫无意义的索引（占总存储消耗的 74%）。虽然在数据库不大的情况下这可能并不是问题，但在大型系统的情况下就不同了——数百 GB 无意义的索引可能会严重伤害总体性能。

可以检查 pg_stat_user_indexes 来找出那些无意义的索引：

```
test=# \d pg_stat_user_indexes
View "pg_catalog.pg_stat_user_indexes"
     Column     |  Type   | Modifiers
---------------+--------+-----------
 relid          | oid    |
 indexrelid     | oid    |
 schemaname     | name   |
 relname        | name   |
 indexrelname   | name   |
 idx_scan       | bigint |
 idx_tup_read   | bigint |
 idx_tup_fetch  | bigint |
```

该视图告诉我们，对于每个方案中每个表上的每个索引，多久会使用它一次（idx_scan）。为了让这个视图丰富一点，笔者建议用下面的 SQL：

```
SELECT schemaname,
       relname,
       indexrelname,
       idx_scan,
       pg_size_pretty(pg_relation_size(indexrelid)),
```

```
        pg_size_pretty(sum(pg_relation_size(indexrelid))
                         OVER (ORDER BY idx_scan, indexrelid)) AS total
FROM    pg_stat_user_indexes
ORDER BY 6 ;
```

这个语句的输出非常有用，它不仅包含有关一个索引多久被使用一次的信息——它还告诉我们为每个索引浪费了多少空间。最后，这个语句在列 6 中加上了所有的空间消耗。用户现在可以仔细检查该表并且重新思考所有那些很少被使用的索引。关于何时删除一个索引很难给出一个通用规则，因此一些手工检查会更有意义。



不要只是盲目地删除索引。在一些情况下，索引未被使用只是因为使用应用的最终用户与预期不同而已。在最终用户改变的情况下（雇佣了一名新的秘书等），一个索引可能又会很好地再次转变成有用的对象。

还有一个名为 pg_statio_user_indexes 的视图，它包含有关索引的缓冲信息。尽管它很有趣，但它通常也不包含能导致重大改进的信息。

5. 跟踪后台工作者

在本节中，是时候看看后台写入器的统计信息了。如你所知，数据库连接在很多情况下不会直接写块到磁盘中。相反，数据由后台写入器进程或者检查点进程写入。

要看看数据如何被写入，可以检查 pg_stat_bgwriter 视图：

```
test=# \d pg_stat_bgwriter
              View "pg_catalog.pg_stat_bgwriter"
         Column         |           Type           | Modifiers
------------------------+--------------------------+-----------
 checkpoints_timed      | bigint                   |
 checkpoints_req        | bigint                   |
 checkpoint_write_time  | double precision         |
 checkpoint_sync_time   | double precision         |
 buffers_checkpoint     | bigint                   |
 buffers_clean          | bigint                   |
 maxwritten_clean       | bigint                   |
 buffers_backend        | bigint                   |
 buffers_backend_fsync  | bigint                   |
 buffers_alloc          | bigint                   |
 stats_reset            | timestamp with time zone |
```

这里首先会吸引读者注意力的事情是前两列。在本书后面的部分读者会学到 PostgreSQL 将执行定期的检查点，它们对于确保数据确实被写到磁盘上是必需的。如果

用户的检查点相互之间离得太近，checkpoint_req 可能会为用户指出正确的方向。如果请求的检查点太多，可能意味着有很多数据被写入并且检查点总是因为高吞吐而被触发。除此之外，PostgreSQL 将告诉用户一个检查点期间写入数据所花的时间以及同步所需的时间。还有，buffers_checkpoint 指示检查点期间有多少缓冲区被写入，以及有多少是被后台写入器写入的（buffers_clean）。

maxwritten_clean 告诉我们后台写入器由于写入了太多缓冲区停止清理扫描的次数。

最后，还有 buffers_backend（由后端数据库连接直接写入的缓冲区数量）、buffers_backend_fsync（一个数据库连接刷写的缓冲区数量）以及 buffers_alloc，它包含被分配的缓冲区数量。

6. 跟踪、归档以及流

在本节中，我们将看到一些复制和事务日志归档相关的特性。第一项要检查的是 pg_stat_archiver，它会告诉我们有关归档进程的信息，归档进程会将事务日志（WAL）从主服务器移动到某种备份设备：

```
test=# \d pg_stat_archiver
            View "pg_catalog.pg_stat_archiver"
       Column        |           Type           | Modifiers
---------------------+--------------------------+-----------
 archived_count      | bigint                   |
 last_archived_wal   | text                     |
 last_archived_time  | timestamp with time zone |
 failed_count        | bigint                   |
 last_failed_wal     | text                     |
 last_failed_time    | timestamp with time zone |
 stats_reset         | timestamp with time zone |
```

pg_stat_archiver 包含有关归档进程的重要信息。首先，它会报告有关已经被归档的事务日志文件的数量（archived_count）。它还知道最后一个被归档的文件以及何时被归档（last_archived_wal 和 last_achived_time）。

虽然了解 WAL 文件的数量很有趣，但这真的不重要。因此可以考虑看一看 failed_count 和 last_failed_wal。如果事务日志归档失败，它将会告诉用户最后一个失败的文件以及是何时发生。推荐对这些域保持关注，因为不这样做就可能出现归档没有工作但用户却没有得到通知的情况。

如果用户正在运行流复制，下面两个视图就对用户非常重要。第一个名为 pg_stat_replication，它提供有关从主机到从机的流进程的信息。每个 WAL 发送器进程都有一项

可见。如果一个项都没有，就没有事务日志流在进行，这可能不是用户想看到的。

让我们看看 pg_stat_replication：

```
test=# \d pg_stat_replication
          View "pg_catalog.pg_stat_replication"
       Column      |           Type           | Modifiers
------------------+--------------------------+-----------
 pid              | integer                  |
 usesysid         | oid                      |
 usename          | name                     |
 application_name | text                     |
 client_addr      | inet                     |
 client_hostname  | text                     |
 client_port      | integer                  |
 backend_start    | timestamp with time zone |
 backend_xmin     | xid                      |
 state            | text                     |
 sent_location    | pg_lsn                   |
 write_location   | pg_lsn                   |
 flush_location   | pg_lsn                   |
 replay_location  | pg_lsn                   |
 sync_priority    | integer                  |
 sync_state       | text                     |
```

在这个视图中将能找到表示流复制连接上的用户名，然后还有一个应用名与连接数据（client_）关联在一起。接着，PostgreSQL 将告诉我们流连接何时启动。在生产系统中，一个年轻的连接可能表明一种网络问题甚至更坏的情况（可靠性问题等）。state 列展示流的其他端处于哪种状态。注意在第 10 章中将会有更多有关于此的信息。

有一些域告诉我们通过网络连接已经发送了多少事务日志（sent_location）、向内核发送了多少事务日志（write_location）、有多少被刷写到磁盘（flush_location）以及有多少已经被重放（replay_location）。最后还列出了同步状态。

pg_stat_replication 可以在复制设置中的发送服务器上查询，而 pg_stat_wal_receiver 则可以在接收端被参考。它能提供类似的信息并且允许在复制品（或是类似的地方）上提取这些信息。

这里是该视图的定义：

```
test=# \d pg_stat_wal_receiver
            View "pg_catalog.pg_stat_wal_receiver"
         Column        |           Type           | Modifiers
-----------------------+--------------------------+-----------
```

```
 pid                   | integer                  |
 status                | text                     |
 receive_start_lsn     | pg_lsn                   |
 receive_start_tli     | integer                  |
 received_lsn          | pg_lsn                   |
 received_tli          | integer                  |
 last_msg_send_time    | timestamp with time zone |
 last_msg_receipt_time | timestamp with time zone |
 latest_end_lsn        | pg_lsn                   |
 latest_end_time       | timestamp with time zone |
 slot_name             | text                     |
 conninfo              | text                     |
```

首先，PostgreSQL 告诉我们 WAL 接收器进程的 ID，然后该视图向我们展示了使用中的连接状态。receive_start_lsn 告诉我们 WAL 接收器被启动时使用的事务日志位置。receive_start_tli 包含 WAL 接收器被启动时在使用的时间线。在某一时刻，用户可能想要知道最后的 WAL 位置及时间线。要得到这两个数字，可使用 received_lsn 和 received_tli。

在接下来的两列中，有两个时间戳：last_msg_send_time 和 last_msg_receipt_time。它们说明一个消息何时被发送以及被收到。

latest_end_lsn 包含在 latest_end_time 时报告给 WAL 发送器进程的最后一个事务日志位置。最后，还有 slot_name 和连接信息的一个含混版本（安全相关的信息被隐去）。

7. 检查 SSL 连接

很多运行 PostgreSQL 的人使用 SSL 来加密从服务器到客户端的连接。近期版本的 PostgreSQL 提供了一个视图来得到那些加密连接的概要，它就是 pg_stat_ssl：

```
test=# \d pg_stat_ssl
   View "pg_catalog.pg_stat_ssl"
   Column    |   Type   | Modifiers
-------------+---------+-----------
 pid         | integer |
 ssl         | boolean |
 version     | text    |
 cipher      | text    |
 bits        | integer |
 compression | boolean |
 clientdn    | text    |
```

每一个进程都由它的进程 ID 表示。如果一个连接使用了 SSL，则第二个列被设置为

真。第三和第四列将定义版本以及密码。最后，还有加密算法使用的位数、是否使用压缩的指示器以及来自客户端证书的**标识名（DN）**域。

8. 实时检查事务

到目前为止，我们已经讨论了几个统计信息表。它们背后的想法都是查看整个系统中在做什么。但是如果用户是一个想要观察个体事务的开发者呢？pg_stat_xact_user_tables 能帮上忙。它不包含系统范围的事务而只有关于当前事务的数据。

```
test=# \d pg_stat_xact_user_tables
View "pg catalog.pg stat xact user tables"
     Column     |  Type  | Modifiers
---------------+--------+-----------
 relid          | oid    |
 schemaname     | name   |
 relname        | name   |
 seq_scan       | bigint |
 seq_tup_read   | bigint |
 idx_scan       | bigint |
 idx tup fetch  | bigint |
 n tup ins      | bigint |
 n tup upd      | bigint |
 n_tup_del      | bigint |
 n_tup_hot_upd  | bigint |
```

因此开发者可以在事务提交前查看该事务，看看它是否导致了性能问题，这样有助于把总体数据与由用户应用导致的数据区分开来。

应用开发者使用这个视图的最理想方式是在事务提交前增加一个函数调用在应用中，跟踪事务做了什么。然后可以检查这些数据，把当前事务的输出与总体负载区分开。

9. 跟踪清理进度

在 PostgreSQL 9.6 中，社区引入了一个很多人一直在期待的系统视图。很多年来，人们都想要跟踪一个清理进程的进度，看看它还有多久才能做完。

pg_stat_progress_vacuum 就是用来回答这类问题的：

```
test=# \d pg stat progress vacuum
View "pg catalog.pg stat progress vacuum"
       Column        |   Type   | Modifiers
--------------------+---------+-----------
 pid                 | integer |
```

```
 datid              | oid     |
 datname            | name    |
 relid              | oid     |
 phase              | text    |
 heap blks total    | bigint  |
 heap blks scanned  | bigint  |
 heap blks vacuumed | bigint  |
 index_vacuum_count | bigint  |
 max_dead_tuples    | bigint  |
 num_dead_tuples    | bigint  |
```

大部分列都是不言而喻的，因此笔者也不再赘述。只有几件事情需要记住。首先，处理不是线性的——它可能会比较跳跃。此外，清理操作通常都很快，因此进展可能很迅速并且难于跟踪。

10. 使用 pg_stat_statements

在讨论了前几个视图之后，现在我们的视线该转回到最重要的视图上，它们可以被用来找出性能问题。笔者说的当然是 pg_stat_statements，其思想是获得有关系统上查询的信息，它可以帮助找出哪种类型的查询很慢以及多久调用一次查询。

要使用这个模块，需要 3 步：

（1）在 postgresql.conf 文件中的 shared_preload_libraries 内加上 pg_stat_statements 。

（2）重启数据库服务器。

（3）在选择的数据库中运行 CREATE EXTENSION pg_stat_statements。

让我们看看该视图的定义：

```
test=# \d pg_stat_statements
          View "public.pg_stat_statements"
        Column        |       Type        | Modifiers
----------------------+-------------------+-----------
 userid               | oid               |
 dbid                 | oid               |
 queryid              | bigint            |
 query                | text              |
 calls                | bigint            |
 total_time           | double precision |
 min_time             | double precision |
 max time             | double precision |
 mean time            | double precision |
 stddev_time          | double precision |
```

```
rows                | bigint           |
shared_blks_hit     | bigint           |
shared_blks_read    | bigint           |
shared_blks_dirtied | bigint           |
shared_blks_written | bigint           |
local_blks_hit      | bigint           |
local_blks_read     | bigint           |
local_blks_dirtied  | bigint           |
local_blks_written  | bigint           |
temp_blks_read      | bigint           |
temp_blks_written   | bigint           |
blk_read_time       | double precision |
blk_write_time      | double precision |
```

pg_stat_statements 提供了极好的信息。对于每个数据库中的每个用户，它为每个查询提供一行。默认情况下它跟踪 5000（可以通过设置 pg_stat_statements.max 来更改）个查询。



查询和参数是分离的。PostgreSQL 将会在查询中放置占位符，这样可以让只是使用不同参数的相同查询被聚集起来。“SELECT ... FROM x WHERE y = 10”将被转变成“SELECT ... FROM x WHERE y = ?”。

对于每个查询，PostgreSQL 会告诉我们它已经消耗的总时间以及调用次数。在近期的版本中，增加了 min_time、max_time、mean_time 和 stddev。标准偏差特别值得一提，因为它将告诉我们一个查询的运行时间是稳定的还是波动的。不稳定的运行时间可能由于多种原因发生：

- 如果查询的数据没有完全被缓存在 RAM 中，查询将去访问磁盘，它们将比数据被缓存时花费更长的时间。
- 不同的参数可能导致不同的计划以及完全不同的结果集。
- 并发和锁可能会有影响。

PostgreSQL 还将告诉我们一个查询的缓冲行为。shared_ columns 列展示有多少块来自于缓存（_hit）或者来自于操作系统（_read）。如果很多块来自于操作系统，查询的运行时间可能会波动。

接下来的一批列都与本地缓冲有关。本地缓冲是由数据库连接直接分配的内存块。

在所有这些信息之上，PostgreSQL 提供了有关临时文件 I/O 的信息。注意在构建大型索引或者执行某些其他大型 DDL 的情况下，临时文件 I/O 将会自然地发生。不过，在 OLTP 场景下临时文件通常都是非常不好的东西，因为它可能造成的磁盘阻塞将会拖慢整个系统。大量的临时文件 I/O 可能表明几种不太好的事情。下面的列表包含了笔者心目中

的前三位：

- 不良的 work_mem 设置（OLTP）。
- 次优的 maintenance_work_mem 设置（DDL）。
- 不应被首先运行的查询。

最后，有两个域包含有关 I/O 计时的信息。默认情况下，这两个域为空。其原因是在某些系统上计时可能本身就会带来很多开销。因此，track_io_timing 的默认值是假——如果需要这类数据记得把它打开。

一旦该模块被启用，PostgreSQL 就开始收集数据，并且用户就可以使用这个视图。



绝不要在一个客户面前运行“SELECT * FROM pg_stat_statements”。笔者不止一次地看到人们会对查询指指点点，他们碰巧知道查询的情况，并且开始解释为什么会这样、谁干的、干了什么、什么时候干的等。在使用这个视图时，总是应该创建一个排序的输出，这样最相关的信息就能马上被看见。

在 Cybertec，我们已经发现下面的查询非常有助于获得数据库服务器行为的总览：

```
test=# SELECT round((100 * total_time/sum(total_time) OVER ())::numeric,
2) percent,
        round(total_time::numeric, 2) AS total,
        calls,
        round(mean_time::numeric, 2) AS mean,
        substring(query, 1, 40)
FROM    pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
 percent |   total    | calls  | mean   |              substring
---------+-----------+--------+-------+--------------------------------
   54.47 | 111289.11 | 122161 |   0.91 | UPDATE pgbench_branches SET
                                           bbalance = b
   43.01 |  87879.25 | 122161 |   0.72 | UPDATE pgbench_tellers SET
                                           tbalance = tb
    1.46 |   2981.06 | 122161 |   0.02 | UPDATE pgbench_accounts SET
                                           abalance = a
    0.50 |   1019.83 | 122161 |   0.01 | SELECT abalance FROM
                                           pgbench_accounts WH
    0.42 |    856.22 | 122161 |   0.01 | INSERT INTO pgbench_history
                                           (tid, bid, a
    0.04 |     85.63 |      1 | 85.63 | copy pgbench_accounts from
                                           stdin
    0.02 |     44.11 |      1 | 44.11 | vacuum analyze pgbench_accounts
```

```
    0.02 |      42.86 | 122161 |   0.00 | END;
    0.02 |      34.08 | 122171 |   0.00 | BEGIN;
    0.01 |      22.46 |        1 | 22.46 | alter table pgbench accounts
                                          add primary
(10 rows)
```

它显示排名前十的查询及其运行时间，包括比例。它还显示该查询的平均执行时间，这样用户可以决定哪些查询的运行时间是否太高。

用户可以用自己的方式检查这个列表并且观察所有看起来平均运行时间过长的查询。

记住检查前 1000 个查询通常并不划算。在大部分情况下，前几个查询就已经对应了系统上的大部分负载。



在笔者的例子中，笔者使用了一个子串来把查询缩短以适合页面。如果想看到实际的查询语句可以不这么做。

记住 pg_stat_statements 将默认把查询在 1024 字节处切断：

```
test=# SHOW track_activity_query_size;
 track_activity_query_size
---------------------------
 1024
(1 row)
```

可以考虑把这个值增加到 16384。如果用户的客户端运行基于 Hibernate 的 Java 应用，一个较大的 track_activity_query_size 值将确保查询不会在有趣的部分被显示之前就被切断。

笔者想在这里指出 pg_stat_statements 到底有多重要，它是目前为止跟踪到性能问题的最容易的方式。一个慢查询日志绝对不如 pg_stat_statements 那么有用，因为慢查询日志只会指出个别的慢查询——它不会告诉我们无数中等查询导致的问题。因此，笔者推荐总是打开这个模块。它的代价其实很小并且不会伤害系统的总体性能。

默认情况下，PostgreSQL 会跟踪（从 9.6 版本起）5000 种查询。在大部分正常合理的应用中，这已经足够。

要重置这些数据，考虑使用下面的指令：

```
test=# SELECT pg stat statements reset();
 pg stat statements reset
--------------------------

(1 row)
```

5.2　创建日志文件

在深入地看过了 PostgreSQL 提供的系统视图之后，现在可以来配置日志了。幸运的是，PostgreSQL 提供了简单的方式来使用日志文件并且帮助人们容易地设置一个好的配置。收集日志很重要，因为它能指出错误和潜在的数据库问题。

所有需要的参数都在 postgresql.conf 文件中。

- 配置 postgresql.conf 文件

本节将介绍 postgresql.conf 文件中一些对于配置日志最重要的项，以及如何以最有利的方式使用日志。

在开始前，笔者想要对 PostgreSQL 中的日志说几句。在 Unix 系统上，PostgreSQL 默认将把日志信息发送到 stderr。不过 stderr 对于日志并不是好地方，因为用户肯定想要在以后的某一时刻检查日志流。因此，利用本章的知识根据需要进行调整就会非常有意义。

1. 定义日志目的地和轮转

让我们先扫一眼 postgresql.conf 文件并看看可以做些什么：

```
#------------------------------------------------------------------
# ERROR REPORTING AND LOGGING
#-----------------------------------------------------------------

# - Where to Log -

#log_destination = 'stderr'
                    # Valid values are combinations of
                    # stderr, csvlog, syslog, and eventlog,
                                 # depending on platform. csvlog
                                 # requires logging_collector to be on.

# This is used when logging to stderr:
#logging_collector = off
                    # Enable capturing of stderr and csvlog
                    # into log files. Required to be on for
                    # csvlogs.
                    # (change requires restart)
```

第一个配置选项定义如何处理日志。日志默认会被送去 stderr（在 Unix 上）。在

Windows 上，日志的默认去向是 eventlog，eventlog 是 Windows 自带的处理日志的工具。用户可以选择将日志送去 csvlog 或者 syslog。

如果用户想要创建 PostgreSQL 日志文件，就应该把去向设置为 stderr 并且打开日志收集器，然后 PostgreSQL 就将创建日志文件。

现在的问题是，那些日志文件的名称是什么以及那些文件被存储在什么地方？postgresql.conf 中就有答案：

```
# These are only used if logging collector is on:
#log_directory = 'pg_log'
                  # directory where log files are written,
                  # can be absolute or relative to PGDATA
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
                  # log file name pattern,
                  # can include strftime() escapes
```

log_directory 会告诉系统在哪里存储日志。如果使用一个绝对路径，用户可以明确地配置日志存在哪里。如果更想让日志文件直接出现在 PostgreSQL 的数据中，可以简单地给一个相对路径。其优势是数据目录将会是自包含的，用户可以移动它而没有后顾之忧。

下一步，用户可以定义 PostgreSQL 要使用的文件名。PostgreSQL 在这一点上非常灵活，允许用户使用 strftime 提供的所有快捷方式。为了让读者对这一特性的强大有些印象，笔者在自己的平台上快速统计了一下，结果显示 strftime 提供了 43（!）种占位符来创建文件名。利用这些占位符，人们的常见需求都可以被实现。

一旦文件名被定义好，很自然地就会考虑清除问题。有下列设置可用：

```
#log_truncate_on_rotation = off
#log_rotation_age = 1d
#log_rotation_size = 10MB
```

默认情况下，如果日志文件存在时间超过一天或者超过 10MB，PostgreSQL 将会持续产生日志文件。log_truncate_on_rotation 指定用户是否想要追加到一个日志文件，因为有时 log_filenames 的定义会让文件名变成循环的，log_truncate_on_rotation 参数规定是要覆盖还是追加到已经存在的文件。对于默认的日志文件设置，这种情况当然不会发生。

一种处理自动轮转的方式是使用类似于 postgresql_%a.log 的命名外加 log_truncate_on_rotation = on。%a 意味着在日志文件名中使用星期几相区分，其优点是这样的文件名每七天就会重复，因此日志文件将在保留一周后被循环使用。但如果目标是每周轮转，10MB 的文件尺寸可能不够，可以考虑关闭最大文件尺寸。

2. 配置 syslog

有些人更愿意使用 syslog 来收集日志文件，PostgreSQL 提供了下列配置参数：

```
# These are relevant when logging to syslog:
#syslog_facility = 'LOCAL0'
#syslog_ident = 'postgres'
#syslog_sequence_numbers = on
#syslog_split_messages = on
```

syslog 在 sysadmins 中间非常流行。幸运的是它很容易配置，基本上用户只需要设置一种日志来源和一个标识符即可。如果 log_destination 被设置为 syslog，默认设置中就已经直接可以让日志收集运转起来了。

3. 记录慢查询

日志也可以被用来跟踪个别慢的查询。以前这几乎就是找出性能问题的唯一方法。

那应该怎么做呢？postgresql.conf 有一个名为 log_min_duration_statement 的变量。如果它被设置为一个大于零的值，每一个超过该设置的查询就会被记入到日志：

```
# log_min_duration_statement = -1
```

大部分人把慢查询日志视为定位性能问题的终极手段。但是，笔者想多加一句提醒，有很多查询确实比较慢，并且它们恰好会吃掉大量 CPU：索引创建，数据导出，数据分析等。

这些长时间运行的查询完全是在预计之中的，而且在很多情况下并非罪恶的根源。其实更多地归咎于很多较短的查询。例如，1000 个查询×500 毫秒比 2 个查询×5 秒更差。在一些情况下慢查询日志可能会产生误导。

尽管如此，也并不意味着它毫无意义——它只是一种信息的来源而不是信息的根源。

4. 定义记录什么以及怎么记录

在看过了一些基本设置后，现在需要决定记录什么。默认情况下，只有错误将被记录。不过，这可能不够。在本节中，读者将学到可以记录什么以及一个日志行长什么样。

默认情况下，PostgreSQL 不会记录有关检查点的信息，下面的设置正好可以改变这种状况：

```
#log_checkpoints = off
```

同样的道理也适用于连接。只要一个连接被建立或者被正确地销毁，PostgreSQL 就

创建日志项：

```
#log_connections = off
#log_disconnections = off
```

在大部分情况下，记录连接没有意义，因为昂贵的记录代价会明显拖慢系统。分析型系统可能不会受到什么影响，但 OLTP 可能会受到严重的影响。

如果用户想要看看语句花了多长时间，可以考虑把下列设置切换为 on：

```
#log_duration = off
```

让我们进入最重要的设置之一。到目前为止，我们还没有确定消息的布局，并且日志文件包含如下形式的错误：

```
test=# SELECT 1 / 0;
ERROR: division by zero
```

日志将会记下 ERROR 以及错误消息，但没有时间戳、用户名等信息。要改变这一状况，可以看看 log_line_prefix：

```
#log_line_prefix = '' # special values:
                      # %a = application name
                      # %u = user name
                      # %d = database name
                      # %r = remote host and port
                      # %h = remote host
                      # %p = process ID
                      # %t = timestamp without milliseconds
                      # %m = timestamp with milliseconds
                      # %n = timestamp with milliseconds (as a
                        Unix epoch)
                      # %i = command tag
                      # %e = SQL state
                      # %c = session ID
                      # %l = session line number
                      # %s = session start timestamp
                      # %v = virtual transaction ID
                      # %x = transaction ID (0 if none)
                      # %q = stop here in non-session processes
                      # %% = '%'
```

log_line_prefix 非常灵活并且允许用户配置日志行来匹配需求。通常，记录时间戳是

一个好主意。否则，基本上无法知道不好的事情是何时发生的。笔者个人还喜欢了解用户名、事务 ID 和数据库。不过还是取决于用户实际需要什么。

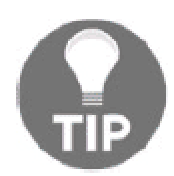
有时速度慢是不好的锁行为造成的。通常来说锁相关的问题很难追踪，log_lock_waits 有助于检测到这种问题。如果打开下面的配置变量，当一个锁被持有的时间超过 deadlock_timeout 时，会有一行被发送到日志：

```
#log_lock_waits = off
```

最后要告诉 PostgreSQL 实际要记录什么。到目前为止，只有错误、慢查询等会被发送到日志。log_statement 有 3 种可能的设置：

```
#log statement = 'none'
# none, ddl, mod, all
```

none 意味着只有错误将被记录。ddl 表示错误以及 DDL（CREATE TABLE、ALTER TABLE 等）会被记录。mod 包括数据更改，而 all 将把每一个语句都发送到日志。



all 可能导致很多日志信息，这可能会拖慢系统。为了让读者了解影响有多大，笔者编写了一篇博文。

如果读者想要检查复制的细节，考虑打开下列设置：

```
#log_replication_commands = off
```

它将把复制相关的命令发送到日志（更多信息请访问下列网站：https://www.postgresql.org/docs/9.6/static/protocol-replication.html）。

由临时文件 I/O 造成的性能问题可能会发生得很频繁。要看看哪些查询导致这类问题，可以使用下列设置：

```
#log_temp_files = -1    # log temporary files equal or larger
                        # than the specified size in kilobytes;
                        # -1 disables, 0 logs all temp files
```

尽管 pg_stat_statements 包含了聚合的信息，log_temp_files 仍能找出导致问题的特定查询。通常这个参数设置为一个合理的较低值，正确的值取决于用户的负载，但是 4MB 可能是一个好的开始。

默认情况下，PostgreSQL 将以服务器所在的时区写日志文件。但是，如果用户运行的是一个遍布全世界的系统，调整出一个用户可以使用并比较日志项的时区更加合理。

```
log_timezone = 'Europe/Vienna'
```

记住在 SQL 方，用户仍将看到用户本地时区的时间。不过，如果设置这个变量，日

志项会在一个不同的时区中。

5.3 总　结

本章全都与系统统计信息有关。用户学到了如何从 PostgreSQL 中抽取信息以及如何以有利的方式使用系统统计信息。本章还详细讨论了最重要的视图。第 6 章将全部与查询优化有关，用户将学习如何检查查询以及如何优化它们。

第 6 章　优化查询获得良好性能

在前几章中，读者已经学到了如何阅读系统统计信息以及如何利用 PostgreSQL 提供的特性。有了这些知识的武装，本章将介绍同良好查询性能有关的内容。读者将进一步学到与下列主题有关的内容：

- 优化器内部。
- 计划执行。
- 分区数据。
- 启用以及禁用优化器设置。
- 与良好查询性能有关的参数。

在本章结束时，笔者希望读者能写出更好和更快的查询，并且如果查询偶然执行得不好时，读者应该能够理解为何出现这样的情况。

6.1　学习优化器的行为

在尝试思考查询性能之前，有理由让读者先熟悉查询优化器的行为。对后台在进行什么工作有更深入的理解是非常有意义的，因为这能帮助读者明白数据库将要做的事情以及正在做的事情。

- 通过例子理解优化

为了展示优化器如何工作，笔者准备了一个已经在 PostgreSQL 培训中使用了多年的例子。假定有 3 个表：

```
CREATE TABLE a (aid int, ...);      -- 1 亿行
CREATE TABLE b (bid int, ...);      -- 2 亿行
CREATE TABLE c (cid int, ...);      -- 3 亿行
```

让我们进一步假设这些表包含数百万或者可能数亿行。此外，其上还有索引：

```
CREATE INDEX idx a ON a (aid);
CREATE INDEX idx_b ON a (bid);
CREATE INDEX idx c ON a (cid);

CREATE VIEW v AS
```

```
        SELECT *
        FROM a, b
        WHERE aid = bid;
```

最后，有一个视图将前两个表连接起来。

让我们假定现在终端用户想要运行下面的查询。优化器将对这个查询做什么，有哪些选择？

```
SELECT  *
FROM    v, c
WHERE   v.aid = c.cid
        AND cid = 4;
```

在着眼于真实的执行处理之前，笔者想要先关注一些规划器拥有的选项。

1. 评估连接选项

在这里规划器有若干种选项，笔者想要借此机会展示如果使用了没有价值的方法，会出现什么问题。

假定规划器只是埋头苦干并且计算该视图的输出，那么连接 1 亿行和 2 亿行的最佳方法是什么？

在本节中，将讨论若干（并不是全部）连接选项来展示 PostgreSQL 可以做些什么。

（1）嵌套循环

一种连接两个表的方式是使用一个嵌套循环。原理非常简单，这里有一些伪代码：

```
for x in table1:
   for y in table2:
         if x.field == y.field
               issue row
         else
               keep doing
```

如果连接的一端非常小并且只包含有限的数据集合，常常会使用嵌套循环。在我们的例子中，一个嵌套循环会导致 1 亿×2 亿次迭代。这显然不是一种好的选项，因为运行时间会暴涨。

嵌套循环的复杂度通常是 $O(n^2)$，因此只有当连接的一端非常小时嵌套循环才有效。在笔者的例子中，情况当然不是这样，因此可以从计算该视图的选项中划掉嵌套循环。

（2）哈希连接

第二种选项是哈希连接。下面的策略可以解决我们的一点小问题：

```
← 哈希连接
        ← 顺序扫描表 1
        ← 顺序扫描表 2
```

连接的两端都可以被哈希并且可以通过比较哈希键来得到连接的结果。这里的麻烦是所有的值都必须被哈希并且放在某个地方。

（3）归并连接

最后，还有归并连接。它的思想是使用排序过的列表来连接结果。如果连接的两端都是有序的，系统可以只是从顶部拿出行，看看它们是否匹配并且返回它们。这里的主要要求是列表要是有序的。下面有一个计划的例子：

```
← 归并连接
        ← 排序表 1
                ← 顺序扫描表 1
        ← 排序表 2
                ← 顺序扫描表 2
```

要进行连接，数据必须以排好序的顺序提供。在很多情况下，PostgreSQL 将会排序数据。不过，还有其他的选项能为连接提供有序的数据。其中之一是参考一个索引，如下例所示：

```
← 归并连接
        ← 索引扫描表 1
        ← 排序表 2
                ← 顺序扫描表 2
```

连接的一端或者两端可以使用来自于计划较低层的有序数据。如果表被直接访问，索引是一种显而易见的选择。

归并连接的妙处在于它能处理大量数据，而其缺点是数据必须被排序或者在某个时刻从索引中取得。

排序的复杂度是 $O(n*\log(n))$。因此，排序 3 亿行数据来执行这一查询也没有什么吸引力。

2. 应用转换

显然，做那些显而易见的事情（先连接视图）根本没有意义。嵌套循环会让执行时间飞涨，哈希连接必须哈希数百万行而归并连接[①]必须排序 3 亿行。所有 3 种选项显然都

[①] 原文为“nested loop”，应为笔误。

不适合这里的情况，比较好的出路是应用逻辑转换让查询变快。在本节中，读者将学到规划器怎样来加速这一查询。若干个步骤会被展示。

（1）内联视图

优化器做的第一个转换是内联视图。就像这样：

```
SELECT   *
FROM     (SELECT *
         FROM a, b
         WHERE aid = bid ) AS v, c
WHERE    v.aid = c.cid
         AND cid = 4;
```

该视图被内联到查询中并且转换成一个子查询。这会为我们带来什么？实际上，什么也没有。它所做的一切只是为进一步的优化打开了一扇门，优化实际上才是这个查询的游戏规则改变者。

（2）扁平化子查询

下一件事情是扁平化子查询。去掉子查询，更多优化查询的选项会出现。

下面是扁平化子查询之后查询的样子：

```
SELECT   *
FROM     a, b, c
WHERE    a.aid = c.cid
         AND aid = bid
         AND cid = 4;
```

现在它是一个普通的连接。注意，虽然我们可以自己完成这一切，但是规划器会帮我们搞定那些转换。现在通往关键优化的大门已经打开。

3. 应用等值约束

下面的处理会创建等值约束。其想法是检测额外的约束、连接选项和过滤条件。让我们深呼吸并且看看这个查询：如果 aid = cid 且 aid = bid，我们知道有 bid = cid。如果 cid = 4 且所有其他列也相等，我们就知道 aid 和 bid 也必须是 4，这就把我们导向了下面的查询：

```
SELECT   *
FROM     a, b, c
WHERE    a.aid = c.cid
         AND aid = bid
         AND cid = 4
```

```
          AND bid = cid
          AND aid = 4
          AND bid = 4
```

这个优化的重要性再怎么强调也不为过。规划器在这里做的是为两个额外的索引创造了机会，这两个索引在原始查询中显然是用不到的。

能够在所有 3 列上使用索引，就没有必要再去计算这个贵得可怕的视图。PostgreSQL 有办法只是从索引检索到若干行，并且使用任意合乎情理的连接选项。

4. 穷举搜索

现在那些形式的转换已经完成，PostgreSQL 将执行一次穷举搜索。它将尝试所有可能的计划并且将最便宜的方案交给查询。PostgreSQL 知道哪些索引是可能的并且使用代价模型来决定如何以最好方法来执行。

在穷举搜索期间，PostgreSQL 还将尝试决定最好的连接顺序。在原始查询中，连接顺序被固定为 A→B 和 A→C。但是，使用那些等值约束我们可以连接 B→C 并且之后再连接 A。所有的选项都在规划器的考虑范围内。

在进行连接时，一般的规则是在处理中早早地先去掉尽可能多的数据。如果这些数据被去掉，后面它就不会再成为噩梦并且降低速度。

5. 全都试一遍

现在所有这些优化都已经被讨论过了，是时候看看 PostgreSQL 能为我们创建什么样的计划了：

```
test=# explain SELECT *
FROM v, c
WHERE v.aid = c.cid
AND cid = 4;
                              QUERY PLAN
----------------------------------------------------------------
 Nested Loop (cost=1.71..17.78 rows=1 width=12)
   -> Nested Loop (cost=1.14..9.18 rows=1 width=8)
       -> Index Only Scan using idx_a on a
            (cost=0.57..4.58 rows=1 width=4)
            Index Cond: (aid = 4)
       -> Index Only Scan using idx_b on b
            (cost=0.57..4.59 rows=1 width=4)
            Index Cond: (bid = 4)
   -> Index Only Scan using idx_c on c
```

```
              (cost=0.57..8.59 rows=1 width=4)
         Index Cond: (cid = 4)
(8 rows)
```

如你所见，PostgreSQL 将使用 3 个索引。有趣的是，PostgreSQL 决定使用一个嵌套循环来连接数据。这是很有意义的，因为几乎没有数据从索引扫描中产生。因此，使用一个循环来进行连接完全可行并且非常高效的。

6. 让处理失败

到目前为止，读者已经看到了 PostgreSQL 可以为我们做什么以及优化器如何帮助加速查询。PostgreSQL 相当聪明，但它仍然需要聪明的用户。在一些情况中，最终用户可能做一些愚蠢的事情毁掉整个优化处理。让我们删除该视图：

```
test=# DROP VIEW v;
DROP VIEW
```

现在重建视图。注意在视图的末尾加上了“OFFSET 0”：

```
test=# CREATE VIEW v AS
SELECT *
FROM a, b
WHERE aid = bid
OFFSET 0;
CREATE VIEW
```

虽然这个视图逻辑上等效于之前展示的例子，但优化器会以不同的方式对待它。每个非 0 的 OFFSET 都将改变结果，因此该视图必须被计算。整个优化处理会被增加的如 OFFSET 之流严重削弱。



PostgreSQL 社区不敢优化在视图中放一个“OFFSET 0”这种愚蠢的情况，人们根本不会那样做。笔者在这里只是把它用作一个例子来展示一些操作可能会削弱性能，并且开发者应该意识到底层的优化处理。

这里是新的计划：

```
test=# EXPLAIN SELECT *
FROM v, c
WHERE v.aid = c.cid
AND cid = 4;
                         QUERY PLAN
-----------------------------------------------------------------
```

```
 Nested Loop (cost=120.71..7949879.40 rows=1 width=12)
   -> Subquery Scan on v
            (cost=120.13..7949874.80 rows=1 width=8)
        Filter: (v.aid = 4)
        -> Merge Join (cost=120.13..6699874.80
                       rows=100000000 width=8)
             Merge Cond: (a.aid = b.bid)
             -> Index Only Scan using idx_a on a
                  (cost=0.57..2596776.57 rows=100000000
                   width=4)
             -> Index Only Scan using idx_b on b
                  (cost=0.57..5193532.33 rows=199999984
                   width=4)
   -> Index Only Scan using idx c on c
        (cost=0.57..4.59 rows=1 width=4)
        Index Cond: (cid = 4)
(9 rows)
```

只需要看一下规划器预测的代价，代价像坐火箭一样从一个两位数飙升到了一个令人震惊的数字。很明显，这个查询将会为用户提供不好的性能。

有更多的方式可以削弱性能，因此牢记优化过程是有意义的。

7. 常量折叠

不过，在 PostgreSQL 中有更多的优化在后台进行并且为总体的好性能做出贡献。这些特性之一叫作**常量折叠**。其思想是把表达式转化成常量，如下例所示：

```
test=# explain SELECT * FROM a WHERE aid = 3 + 1;
                          QUERY PLAN
-----------------------------------------------------------------
 Index Only Scan using idx a on a
         (cost=0.57..4.58 rows=1 width=4)
   Index Cond: (aid = 4)
(2 rows)
```

如你所见，PostgreSQL 将尝试查找 4。由于 aid 上有索引，PostgreSQL 将会进行一次索引扫描。注意我们的表只有一列，因此 PostgreSQL 甚至能在索引中找到它所需要的所有数据。

如果该表达式在左手边会发生什么？

```
test=# explain SELECT * FROM a WHERE aid - 1 = 3;
                         QUERY PLAN
```

```
-------------------------------------------------------------
 Seq Scan on a (cost=0.00..1942478.48 rows=500000 width=4)
   Filter: ((aid - 1) = 3)
(2 rows)
```

在这种情况下，索引查找会失败并且 PostgreSQL 只能求助于一个顺序扫描。

8. 理解函数内联

正如本节中已经提到的，有很多优化可以帮助加速查询，其中之一被称作**函数内联**。PostgreSQL 能够内联不可变 SQL 函数，主要的思想是减少必须要做的函数的调用次数以提高速度。

这里是一个函数的例子：

```
test=# CREATE OR REPLACE FUNCTION ld(int)
RETURNS numeric AS
$$
   SELECT log(2, $1);
$$
LANGUAGE 'sql' IMMUTABLE;
CREATE FUNCTION
```

该函数将计算输入值的对数二元（logarithmus dualis）：

```
test=# SELECT ld(1024);
         ld
---------------------
 10.0000000000000000
(1 row)
```

为了展示工作原理，笔者将用较少的内容重建该表以便加速索引创建：

```
test=# TRUNCATE a;
TRUNCATE TABLE
```

然后可以再次加入数据并且创建索引：

```
test=# INSERT INTO a SELECT * FROM generate_series(1, 10000);
INSERT 0 10000
test=# CREATE INDEX idx_ld ON a (ld(aid));
CREATE INDEX
```

不出所料，在该函数上创建的索引就像任何其他索引一样派上了用场。但是，仔细看看索引条件：

```
test=# EXPLAIN SELECT * FROM a WHERE ld(aid) = 10;
                          QUERY PLAN
-----------------------------------------------------------------
 Index Scan using idx ld on a (cost=0.29..8.30 rows=1 width=4)
   Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 rows)
```

这里的重要观察是，索引条件实际查找的是 log 函数而不是 ld 函数。优化器完全去掉了那一次函数调用。

逻辑上来讲，这为下面的查询提供了机会：

```
test=# EXPLAIN SELECT * FROM a WHERE log(2, aid) = 10;
                           QUERY PLAN
------------------------------------------------------------------
 Index Scan using idx_ld on a (cost=0.29..8.30 rows=1 width=4)
   Index Cond: (log('2'::numeric, (aid)::numeric) = '10'::numeric)
(2 rows)
```

9. 连接剪枝

PostgreSQL 提供一种名为**连接剪枝**的优化，其思想是移除查询不需要的连接。这在查询由某种中间件或者 ORM 生成的场景中派得上用场。如果一个连接可以被移除，这自然会显著地提高速度并且导致较少的开销。

现在的问题是：连接剪枝如何工作？这里有一个例子：

```
CREATE TABLE x (id int, PRIMARY KEY (id));
CREATE TABLE y (id int, PRIMARY KEY (id));
```

首先，两个表被创建，确保连接条件的两端实际上都是唯一的。那些约束一会儿会非常重要。

现在可以写一个简单的查询：

```
test=# EXPLAIN SELECT *
FROM x LEFT JOIN y
ON (x.id = y.id)
WHERE x.id = 3;
                           QUERY PLAN
-----------------------------------------------------------------
 Nested Loop Left Join (cost=0.31..16.36 rows=1 width=8)
   Join Filter: (x.id = y.id)
   -> Index Only Scan using x_pkey on x
```

```
        (cost=0.15..8.17 rows=1 width=4)
        Index Cond: (id = 3)
   -> Index Only Scan using y pkey on y
        (cost=0.15..8.17 rows=1 width=4)
        Index Cond: (id = 3)
(6 rows)
```

如你所见，PostgreSQL 直接连接这些表，目前为止没有意外出现。不过，下面的查询有一点修改。它只选择那些位于连接左手边的列，而不是所有列：

```
test=# explain SELECT x.*
FROM x LEFT JOIN y
ON (x.id = y.id)
WHERE x.id = 3;
                           QUERY PLAN
----------------------------------------------------------------
 Index Only Scan using x pkey on x
   (cost=0.15..8.17 rows=1 width=4)
   Index Cond: (id = 3)
(2 rows)
```

PostgreSQL 将进行一个直接的内侧扫描，并且完全跳过连接。能够这样做且逻辑上正确的原因有两点：

- 没有选择从连接的右边出列，因此查看那些列不会为我们带来任何好处。
- 右侧是唯一的，这意味着连接不会由于右侧的重复而增加行数。

如果连接能被自动地剪枝，查询的速度可能会提高一个量级。这里的妙处在于，只是通过移除应用不需要的列就能实现提速。

10. 加速集合操作

集合操作允许多个查询的结果被组合成一个结果集，集合操作符包括 UNION、INTERSECT 和 EXCEPT。PostgreSQL 实现了所有这几种操作符，并且提供了很多重要的优化来为它们加速。

规划器能够把限制下推到集合操作，这就为精巧的索引和一般的加速普遍提供了机会。让我们看看下面的查询，它会展示其中的道理：

```
test=# EXPLAIN SELECT *
FROM (
SELECT aid AS xid
FROM   a
UNION ALL
```

```
SELECT   bid
FROM b
) AS y
WHERE xid = 3;
                          QUERY PLAN
-----------------------------------------------------------------
 Append (cost=0.29..12.89 rows=2 width=4)
   -> Index Only Scan using idx_a on a
        (cost=0.29..8.30 rows=1 width=4)
        Index Cond: (aid = 3)
   -> Index Only Scan using idx_b on b
        (cost=0.57..4.59 rows=1 width=4)
        Index Cond: (bid = 3)
(5 rows)
```

从中可以看到，两个关系被相互加在一起。问题是唯一的限制在子查询之外。不过，PostgreSQL 发现这个过滤条件能在计划中进一步被下推。因此，xid = 3 被附在 aid 和 bid 上，为在两个表上使用索引打开了选项。通过避免两个表上的顺序扫描，该查询将会运行得快很多。

注意，UNION 子句和 UNION ALL 子句有一点区别。UNION ALL 子句只是盲目地追加数据并且传递出两个表的结果。而 UNION 子句不同，它将过滤掉重复。下面的计划展示了其工作原理：

```
test=# EXPLAIN SELECT *
FROM (
SELECT  aid AS xid
FROM    a
UNION
SELECT  bid
FROM b
) AS y
WHERE xid = 3;
                          QUERY PLAN
-----------------------------------------------------------------
 Unique (cost=12.92..12.93 rows=2 width=4)
   -> Sort (cost=12.92..12.93 rows=2 width=4)
        Sort Key: a.aid
        -> Append (cost=0.29..12.91 rows=2 width=4)
             -> Index Only Scan using idx_a on a
                  (cost=0.29..8.30 rows=1 width=4)
                  Index Cond: (aid = 3)
```

```
              -> Index Only Scan using idx_b on b
                   (cost=0.57..4.59 rows=1 width=4)
                   Index Cond: (bid = 3)
(8 rows)
```

PostgreSQL 不得不在 Append 节点的上面增加一个 Sort 节点来确保后面可以过滤掉重复。



很多没有完全认识到 UNION 子句和 UNION ALL 子句区别的人会抱怨性能不好，因为他们没有意识到 PostgreSQL 必须过滤掉重复，这种操作在大型数据集的情况下是特别痛苦的。

6.2　理解执行计划

在钻研了一些 PostgreSQL 实现的重要优化之后，笔者想把读者的注意力更多地转向执行计划。读者已经在本书中见过一些计划。不过，为了完整地利用计划，形成一种阅读这种信息的系统方法很重要，而这正好就是本节要讨论的范畴。

6.2.1　系统地处理计划

第一件要了解的是，EXPLAIN 子句可以做很多工作，笔者强烈建议用户充分利用这些特性。

正如很多读者已经知道的，一个 EXPLAIN ANALYZE 子句将执行查询，并且返回包括实际运行时信息的计划。这里有一个例子：

```
test=# EXPLAIN ANALYZE SELECT *
   FROM (SELECT *
         FROM b
         LIMIT 1000000
         ) AS b
   ORDER BY cos(bid);
                              QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=146173.12..148673.12 rows=1000000)
        (actual time=837.049..1031.587 rows=1000000)
        Sort Key: (cos((b.bid)::double precision))
        Sort Method: external merge Disk: 25408kB
   -> Subquery Scan on b
        (cost=0.00..29424.78 rows=1000000 width=12)
```

```
        (actual time=0.011..352.717 rows=1000000)
        -> Limit (cost=0.00..14424.78 rows=1000000)
             (actual time=0.008..169.784 rows=1000000)
             -> Seq Scan on b b_1 (cost=0.00..2884955.84
                  rows=199999984 width=4)
                  (actual time=0.008..85.710 rows=1000000)
 Planning time: 0.064 ms
 Execution time: 1159.919 ms
(8 rows)
```

这个计划看起来有点吓人，但也不必恐慌，我们将逐步地理解它。在阅读一个计划时，要确保从内向外进行阅读。在笔者的例子中，执行从 b 上的一个顺序扫描开始。这里实际有两块信息：代价块和实际时间块。虽然代价块包含的是估计，但实际时间块中却是真凭实据，它显示实际的执行时间。在这个例子中，该顺序扫描花费了 85.7 毫秒。

然后数据被传递给 Limit 节点，确保不会有太多数据。注意执行的每个阶段还向我们展示涉及的行数。如你所见，PostgreSQL 只从表的最初取得 1 百万行，Limit 节点会确保这一点。不过，这是要付出代价的，在这一阶段，运行时间已经跳到了 169 毫秒。

最后，数据花了很多时间被排序。在查看该计划时最重要的事情是找到时间到底损失在了什么地方。最好的办法是看看实际时间块，并且尝试找出时间暴涨的地方。在这个例子中，顺序扫描花了一些时间，但它无法被明显地提速。相反可以看到在排序开始时，时间就开始飞涨。

当然，排序可以被加速，本章后面的部分会对此介绍更多。

● 让 EXPLAIN 更详细

在 PostgreSQL 中，EXPLAIN 子句的输出可以被充实一点来为用户提供更多信息。要从计划中尽可能多地榨取信息，考虑把下列选项打开：

```
test=# EXPLAIN (
         analyze true,
         verbose true,
         costs true,
         timing true,
         buffers true)
   SELECT * FROM a ORDER BY random();
                        QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=834.39..859.39 rows=10000 width=12)
        (actual time=6.089..7.199 rows=10000 loops=1)
```

```
   Output: aid, (random())
   Sort Key: (random())
   Sort Method: quicksort Memory: 853kB
   Buffers: shared hit=45
   -> Seq Scan on public.a
        (cost=0.00..170.00 rows=10000 width=12)
        (actual time=0.012..2.625 rows=10000 loops=1)
        Output: aid, random()
        Buffers: shared hit=45
 Planning time: 0.054 ms
 Execution time: 7.992 ms
(10 rows)
```

analyze true 像前面展示的那样实际地执行查询。verbose true 增加更多的信息到该计划（例如列信息等）。costs true 显示有关代价的信息。timing true 同样重要，因为它将为我们提供好的运行时数据，这样我们可以看到时间损失在计划中的哪个部分。最后，还有 buffers true，它可能会非常有启发作用。在笔者的例子中，它显示我们用到了 45 个缓冲区来执行该查询。

6.2.2　发现问题

根据第 5 章中展示的所有信息，已经可以发现若干潜在的性能问题，它们在实际生活中非常重要。

1. 发现运行时间中的变化

在查看一个计划时，用户总是会问自己两个问题：

- EXPLAIN ANALYZE 子句展示的运行时间对于给定查询合乎情理吗？
- 如果该查询很慢，运行时间是在哪里暴涨的？

在笔者的例子中，顺序扫描被估价为 2.625 毫秒。排序在 7.199 毫秒后完成，因此排序用了大约 4.5 毫秒来完成，所以它应该对该查询所需的运行时间负大部分的责任。

在查询的执行时间中寻找暴涨点将会揭示到底在发生什么。根据哪种操作烧掉太多时间，用户必须做出相应的应对。这里不可能给出一般性的建议，只是因为有太多事情可能会导致那些问题。

2. 检查估计值

但是，有一些事情总是要做的：确保估计值和实际的数字相当接近。在某些情况

下，优化器将做出不好的决定，因为估计值由于某种原因大错特错。估计值错误可能由于系统统计信息过时导致，因此运行一个 ANALYZE 子句绝对是解决这类问题最好的出发点。但是，优化器的统计信息大部分由 autovacuum 守护进程照管，因此绝对值得考虑其他导致不良估计值的选项。看看下面的例子：

```
test=# CREATE TABLE t estimate AS
   SELECT * FROM generate series(1, 10000) AS id;
SELECT 10000
```

在载入 10000 行后，优化器统计信息被创建：

```
test=# ANALYZE t estimate;
ANALYZE
```

让我们现在来看看估计值：

```
test=# EXPLAIN ANALYZE SELECT *
         FROM t_estimate
         WHERE cos(id) < 4;
                          QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on t estimate (cost=0.00..220.00 rows=3333 width=4)
         (actual time=0.010..4.006 rows=10000 loops=1)
   Filter: (cos((id)::double precision) < '4'::double precision)
 Planning time: 0.064 ms
 Execution time: 4.701 ms
(4 rows)
```

在很多情况下，PostgreSQL 可能无法正确地处理 WHERE 子句，因为它只有列上的统计信息而没有表达式的统计信息。在这里看到从 WHERE 子句返回的数据被严重地低估了。

当然，还可能发生数据量被高估的情况：

```
test=# EXPLAIN ANALYZE SELECT *
 FROM t estimate
 WHERE cos(id) > 4;
 QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on t_estimate (cost=0.00..220.00 rows=3333 width=4)
 (actual time=3.802..3.802 rows=0 loops=1)
 Filter: (cos((id)::double precision) > '4'::double precision)
 Rows Removed by Filter: 10000
```

```
 Planning time: 0.037 ms
 Execution time: 3.813 ms
(5 rows)
```

如果这类事情发生在计划的深处，该过程将很可能创建一个不好的计划。因此，确保估计值在某个范围内的意义非凡。

幸运的是，有一种方法可以应付这种问题：

```
test=# CREATE INDEX idx cosine ON t estimate (cos(id));
CREATE INDEX
```

创建一个索引将让 PostgreSQL 跟踪该表达式的统计信息：

```
test=# ANALYZE;
ANALYZE
```

除这个计划将确保明显更好的性能之外，它还将修正统计信息——即便不使用索引：

```
test=# EXPLAIN ANALYZE SELECT *
 FROM t estimate
 WHERE cos(id) > 4;
 QUERY PLAN
------------------------------------------------------------------
 Index Scan using idx_cosine on t_estimate
 (cost=0.29..8.30 rows=1 width=4)
 (actual time=0.002..0.002 rows=0 loops=1)
 Index Cond: (cos((id)::double precision)
 > '4'::double precision)
 Planning time: 0.095 ms
 Execution time: 0.011 ms
(4 rows)
```

不过，除呈现在眼前的错误估计之外，还有更多可能。一种常被低估的问题叫作跨列关联。考虑这个涉及两个列的简单例子：

- 20%的人喜欢滑雪。
- 20%的人来自非洲。

如果我们想要对来自非洲的滑雪者计数，数学运算会告诉我们结果是总群体的 0.2×0.2 = 4%。不过，在非洲没有雪而且收入较低，因此实际的结果肯定将会更低。对非洲的观察和对滑雪的观察不是统计独立的。在很多情况下，事实是 PostgreSQL 保留的列统计信息不会跨越超过一列，这就会导致不好的结果。当然，规划器会做很多努力来阻止这类事情发生。但是，它仍然可能是一个问题。

从 PostgreSQL 10.0 开始，很可能在 PostgreSQL 中看到多元统计信息，这将一劳永逸地终结跨列关联问题。

3. 检查缓冲区使用

不过，计划本身并非导致问题的唯一因素。很多情况下，危险隐藏在某个其他层次上。内存和缓冲可能会导致意外的行为，对于那些没有被训练查看本节描述的问题的最终用户来说，这些行为通常很难理解。

这里有一个例子：

```
test=# CREATE TABLE t random AS
         SELECT *
         FROM generate series(1, 10000000) AS id
         ORDER BY random();
SELECT 10000000
test=# ANALYZE t_random ;
ANALYZE
```

笔者已经产生了一个包含 10000000 行的简单表并且创建好了优化器统计信息。下一步，执行一个只检索少量行的简单查询：

```
test=# EXPLAIN (analyze true, buffers true, costs true, timing true)
   SELECT *
   FROM t random
   WHERE id < 1000;
                        QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on t_random (cost=0.00..169248.60 rows=1000 width=4)
   (actual time=1.068..685.410 rows=999 loops=1)
   Filter: (id < 1000)
   Rows Removed by Filter: 9999001
   Buffers: shared hit=2112 read=42136
 Planning time: 0.035 ms
 Execution time: 685.551 ms
(6 rows)
```

在观察数据之前，该查询确保已经执行了两次。当然，也可以在这里使用一个索引。但是，笔者想要点出的是另外一件事情。在笔者的查询中，PostgreSQL 在缓存中找了 2112 个缓冲区，并且从操作系统拿到 42136[1]个缓冲区。现在可能发生两件事：如果足

---

[1] 此处上文的执行计划中 read=42136，原文中这里的 421136 应该是笔误。

够幸运，操作系统会达到若干次缓存命中并且查询会很快；如果文件系统缓存不那么幸运，那些块就必须从磁盘取得。这可能是显而易见的，然而，它可能会导致执行时间的剧烈变化。一个完全在缓存中运行的查询可能比一个需要缓慢从磁盘收集随机块的查询快 100 倍。

让我们尝试使用一个简单的例子勾勒这个问题。假设我们有一个电话系统，它存储了 10000000000 行（在大型电话运营商中并不罕见）。数据以一个非常快的速率流入并且用户想要查询这些数据。如果有 10000000000 行，数据将只能部分放入内存，因此很多东西都将自然地来自于磁盘。

现在我们可以运行一个简单的查询：

```
SELECT * FROM data WHERE phone_number = '+12345678';
```

即便用户正在打电话，他/她的数据也将散布在各处。如果用户结束一次通话然后开始下一次通话，数以千计的人也会做同样的事情，因此用户的两次通话结束于完全相同的 8000 字节块的几率自然近乎于零。只是想象一下同时有 100000 个通话进行的时段。在磁盘上，数据将被随机分布。在用户的电话号码经常出现的情况下，这意味着对每一行至少有一块已经被从磁盘上取出（假定缓冲命中率很低）。假定有 5000 行将被返回。假设用户必须去访问磁盘 5000 次，这将导致 5000×5 毫秒 = 25 秒的执行时间。注意这一查询的执行时间可能会在数毫秒到 30 秒之间变化，取决于有多少数据被操作系统或者 PostgreSQL 所缓存。

记住每一次服务器重启将清空 PostgreSQL 和文件系统缓存，这将导致节点失效后的大麻烦。

4. 修正高缓冲区使用

现在的问题是，怎样才能改进这种状况？一种方法是运行 CLUSTER 子句：

```
test=# \h CLUSTER
Command:     CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

CLUSTER 子句将按照一个（B 树）索引的相同顺序重写表。如果正在运行一种分析型负载，这种做法会有意义。不过，在一个 OLTP 系统中，CLUSTER 子句可能就行不通，因为在该表被重写期间要求一个表锁。

6.3　理解并且固定连接

连接是一种重要的操作，每个人都经常会需要用到它们。因此，连接也与维护或达到良好性能相关。为了确保能写出好的连接，笔者决定在本书中用一节来介绍连接。

6.3.1　正确使用连接

在深入优化连接之前，务必要看一些与连接同时出现的最常见的问题，它们应该会对用户敲起警钟。

这里有一个例子：

```
test=# CREATE TABLE a (aid int);
CREATE TABLE
test=# CREATE TABLE b (bid int);
CREATE TABLE
test=# INSERT INTO a VALUES (1), (2), (3);
INSERT 0 3
test=# INSERT INTO b VALUES (2), (3), (4);
INSERT 0 3
```

在下一个例子中，读者将看到一个简单的外连接：

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid);
 aid | bid
-----+-----
 1   |
 2   |   2
 3   |   3
(3 rows)
```

可以看到 PostgreSQL 将从左手边拿出所有行，并且只列出符合条件的行。

接下来的例子可能会让很多人感到惊奇：

```
test=# SELECT * FROM a LEFT JOIN b ON (aid = bid AND bid = 2);
 aid | bid
-----+-----
 1   |
 2   |   2
 3   |
(3 rows)
```

是的，行数没有减少——它仍然保持一个常数。大部分人会设想在连接中将只有一行，但并不是这样并且将导致某些隐藏的问题。

考虑下列查询：

```
test=# SELECT avg(aid), avg(bid)
   FROM a LEFT JOIN b
         ON (aid = bid AND bid = 2);
        avg          |         avg
--------------------+--------------------
 2.0000000000000000 | 2.0000000000000000
(1 row)
```

大部分人会设想这个均值是基于一个单个行计算的。但是，正如前面一点所说的，事实并非如此，因而类似这样的查询常常被认为是一种性能问题。由于某种原因，PostgreSQL 没有对连接左边的表做索引。当然，我们这里看到的并不是一种性能问题——它无疑是一种语义问题。人们书写的外连接与想让 PostgreSQL 做的事情不符的情况经常发生。因此，笔者的个人建议是在解决客户报告的性能问题之前，总是先质疑一下外连接的语义正确性。

这种工作对于确保查询正确非常重要，笔者觉得再怎么强调都不为过。

6.3.2　处理外连接

在从业务的角度验证了查询确实正确之后，就可以检查优化器能做些什么来加速外连接。最重要的事情是，PostgreSQL 在很多情况下通过重排序内连接来显著地提速。但是，在外连接的情况下，并不是总能这样。实际上只允许少数重排序操作的 Pac：

(A leftjoin B on (Pab)) innerjoin C on (Pac)=(A innerjoin C on (Pac)) leftjoin B on (Pab)

Pac 是一个引用 A 和 C 的谓词，诸如此类（在这个例子中，显然 Pac 不能引用 B，否则转换就是没有意义的）：

(A leftjoin B on (Pab)) leftjoin C on (Pac) = (A leftjoin C on (Pac)) leftjoin B on (Pab)

(A leftjoin B on (Pab)) leftjoin C on (Pbc) = (A leftjoin (B leftjoin C on (Pbc)) on (Pab)

只有谓词 Pbc 对所有为空的 B 行都必然失败（即 Pbc 对 B 的至少一列是严格的）。如果 Pbc 不严格，第一种形式可能会产生一些有非空 C 列的行，而第二种形式会让这些项为空。

虽然一些连接可以被重排序，但有一种典型的查询无法从连接重排序获益：

```
SELECT ...
FROM     a LEFT JOIN b ON (aid = bid)
   LEFT JOIN c ON (bid = cid)
```

```
    LEFT JOIN d ON (cid = did)
    ...
```

其解决之道是检查是否所有的外连接真的都是必需的。很多情况下，人们会在不需要外连接的地方写上外连接。通常，商业案例甚至没有使用外连接的必要性。

6.3.3　理解 join_collapse_limit 变量

在规划处理期间，PostgreSQL 会尝试检查所有可能的连接顺序。很多情况下，这个过程可能会相当昂贵，因为可能会有多种排列，它们自然拖慢规划处理。join_collapse_limit 变量给了开发者一种工具来实际地解决这些问题，并且以更直接的方式定义了一个查询应该如何被处理。

为了展示有关这一设置的一切，笔者已经编好了一个小例子：

```
SELECT   *
FROM     tab1, tab2, tab3
WHERE    tab1.id = tab2.id
         AND tab2.ref = tab3.id;
SELECT   *
FROM     tab1 CROSS JOIN tab2
         CROSS JOIN tab3
WHERE    tab1.id = tab2.id
         AND tab2.ref = tab3.id;
SELECT   *
FROM     tab1 JOIN (tab2 JOIN tab3
              ON (tab2.ref = tab3.id))
         ON (tab1.id = tab2.id);
```

说穿了，这 3 个查询都是相同的并且会被规划器以同样的方式对待。第一个查询由隐式连接组成。最后一个查询只由显式连接组成。在内部，规划器将检查那些请求并且相应地排序连接以确保最好的运行时间。现在的问题是，PostgreSQL 会暗中规划多少显式连接？这正是可以通过设置 join_collapse_limit 变量来告诉规划器的信息，其默认值对于普通查询已经相当好了。不过，如果用户的查询包含数量很多的连接，使用这个设置可以可观地减少规划时间。减少规划时间对于维护好的吞吐可能是至关重要的。

为了展示 join_collapse_limit 变量如何改变计划，笔者编写了一个简单查询：

```
test=# EXPLAIN WITH x AS (SELECT * FROM generate series(1, 1000) AS id)
        SELECT   *
        FROM     x AS a JOIN x AS b ON (a.id = b.id)
                 JOIN x AS c ON (b.id = c.id)
```

```
                JOIN x AS d ON (c.id = d.id)
                JOIN x AS e ON (d.id = e.id)
                JOIN x AS f ON (e.id = f.id);
```

尝试用不同的设置运行该查询，然后看看计划如何变化。不幸的是，该计划太长以至于无法把它放在这里，因此笔者无法在本节中展现实际的改变。

6.4　启用和禁用优化器设置

到目前为止，规划器施行的大部分重要优化已经就或多或少的细节进行了讨论。PostgreSQL 这些年来变得越来越聪明。但仍有可能出现事情变得越来越糟的情况，而用户则不得不说服规划器来做正确的事情。

为了修改计划，PostgreSQL 提供了若干运行时变量，它们将对规划产生显著的影响。其思想是让最终用户有机会使得计划中的特定类型节点比其他节点更昂贵。在实践中这意味着什么呢？

这里有一个简单的计划：

```
test=# explain  SELECT *
       FROM     generate series(1, 100) AS a,
                generate series(1, 100) AS b
       WHERE a = b;
                          QUERY PLAN
-----------------------------------------------------------------
 Merge Join (cost=119.66..199.66 rows=5000 width=8)
   Merge Cond: (a.a = b.b)
   -> Sort (cost=59.83..62.33 rows=1000 width=4)
        Sort Key: a.a
        -> Function Scan on generate series a
        (cost=0.00..10.00 rows=1000 width=4)
   -> Sort (cost=59.83..62.33 rows=1000 width=4)
        Sort Key: b.b
        -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(8 rows)
```

该计划显示 PostgreSQL 从函数读取数据，并且排序两个结果，然后执行一个归并连接。

不过，如果归并连接不是运行该查询最快的方法会怎样呢？在 PostgreSQL 中没有办法像 Oracle 那样把规划器提示放在注释中。不过，我们可以确保特定的查询会被认为非

常昂贵。SET enable_mergejoin TO off 命令将会让归并连接过于昂贵：

```
test=# SET enable_mergejoin TO off;
SET
test=# explain  SELECT *
        FROM    generate_series(1, 100) AS a,
                generate_series(1, 100) AS b
        WHERE   a = b;
                          QUERY PLAN
-----------------------------------------------------------------
 Hash Join (cost=22.50..210.00 rows=5000 width=8)
   Hash Cond: (a.a = b.b)
   -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
   -> Hash (cost=10.00..10.00 rows=1000 width=4)
        -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(5 rows)
```

由于归并过于昂贵，PostgreSQL 决定尝试哈希连接。如你所见，代价变高了一点，但由于归并已不在考虑之列，所以仍然采用了这个计划。

如果哈希连接也被关闭会发生什么？

```
test=# SET enable hashjoin TO off;
SET
test=# explain  SELECT *
        FROM    generate_series(1, 100) AS a,
                generate_series(1, 100) AS b
        WHERE   a = b;
                          QUERY PLAN
-----------------------------------------------------------------
 Nested Loop (cost=0.01..22510.01 rows=5000 width=8)
   Join Filter: (a.a = b.b)
   -> Function Scan on generate_series a
        (cost=0.00..10.00 rows=1000 width=4)
   -> Function Scan on generate_series b
        (cost=0.00..10.00 rows=1000 width=4)
(4 rows)
```

PostgreSQL 将再次尝试其他的东西，并且最后用到了嵌套循环。嵌套循环的代价已经令人震惊，但是规划器已别无选择。

如果嵌套循环再被关闭会发生什么？

```
test=# SET enable_nestloop TO off;
SET
test=# explain  SELECT *
        FROM    generate series(1, 100) AS a,
                generate series(1, 100) AS b
        WHERE a = b;
                       QUERY PLAN
-----------------------------------------------------------------
 Nested Loop (cost=10000000000.00..10000022510.00
              rows=5000 width=8)
   Join Filter: (a.a = b.b)
   -> Function Scan on generate series a
        (cost=0.00..10.00 rows=1000 width=4)
   -> Function Scan on generate series b
        (cost=0.00..10.00 rows=1000 width=4)
(4 rows)
```

PostgreSQL 仍将执行嵌套循环。这里的重点是这种“关闭”并非真正意义上的关闭——它只是意味着计划节点会被当作一种非常昂贵的东西。这一点非常重要，否则查询将无法被执行。

有哪些设置可以影响规划器？有下列开关可用：

- enable_bitmapscan = on
- enable_hashagg = on
- enable_hashjoin = on
- enable_indexscan = on
- enable_indexonlyscan = on
- enable_material = on
- enable_mergejoin = on
- enable_nestloop = on
- enable_seqscan = on
- enable_sort = on
- enable_tidscan = on

虽然这些设置绝对是有益的，但笔者想要指出的是，这些调整应该被非常非常小心地处理。只用它们来加速个别查询而不要在全局上去打开。这些设置的效果可能会很快就发生反转并且毁掉性能。因此，在改变这些参数之前真的有必要三思。

● 理解遗传查询优化

规划处理的结果是实现优秀性能的关键。如本章中所示，规划不是一件简单的事情，并且涉及多种复杂的计算。一个查询触及的表越多，规划过程就会变得越复杂。有越多的表，规划器拥有的选择也就更多，逻辑上规划时间将会增加。在某个点上，规划将会花费非常久，以至于执行经典的穷举搜索变得不再可行。除此之外，规划时产生的错误也非常巨大，总之寻找理论上的最优计划并不一定会得到运行时间上的最优计划。

此时**遗传查询优化（GEQO）**可以帮上忙。什么是 GEQO？其思想实际是从自然界借用过来的，它类似于自然进化过程。

PostgreSQL 用类似于旅行商问题的方法来处理 GEQO 问题，并且把可能的连接编码为整数串。例如，4-1-3-2 意味着先连接 4 和 1，然后连接 3，再连接 2，这些数字表示关系的 ID。在最开始，遗传优化器将产生计划的一个随机集合，然后检查那些计划。不好的计划会被放弃并且会基于好计划的基因生成新的计划，这种方式可能会生成更好的计划。这种处理可以被重复必要的次数。到了最后，将会得到一个计划，它预期会比使用随机计划要好得多。

通过调整 geqo 变量可以打开和关闭 GEQO：

```
test=# SHOW geqo;
 geqo
------
 on
(1 row)

test=# SET geqo TO off;
SET
```

默认情况下，如果一个语句超过一定的复杂度，geqo 就会开始起作用，这个复杂度由下面的变量控制：

```
test=# SHOW geqo_threshold ;
 geqo_threshold
----------------
 12
(1 row)
```

如果用户的查询大到开始达到这一阈值，当然就应该尝试一下这个设置来看看更改变量时规划器会怎样改变计划。

不过，一般来说，笔者建议应该尽量避免使用 GEQO 并且尝试通过使用 join_collapse_

limit 变量固定连接顺序来解决问题。注意每一个查询都是不同的，因此做实验和通过学习规划器在各种情况下的行为获得经验是很有帮助的。



如果读者想看看真正疯狂的连接，可以考虑看看笔者在 Madrid 的谈话。

6.5　分 区 数 据

给定默认的 8000 字节块，PostgreSQL 可以在单表内存储多达 32TB 的数据。如果用户用 32000 字节块编译 PostgreSQL，甚至可以在单表内放进多达 128TB。但是，那么大的表并不一定很方便，因此有必要对表分区来方便处理，更方便并且在某些情况下更快速。

从 PostgreSQL 10.0 开始，我们将很可能用上改进过的分区特性，它将为最终用户提供更方便的数据分区处理。

在写作本章时，PostgreSQL 10.0 还没有被发布，因此本章介绍的还是旧的分区方式。

6.5.1　创建分区

在深入分区的优点之前，笔者想要展示如何创建分区。整件事情开始于一个父表：

```
test=# CREATE TABLE t_data (id serial, t date, payload text);
CREATE TABLE
```

在这个例子中，父表有 3 列。date 列将被用作分区，这个稍后会谈到更多。

现在父表已经就位，可以创建子表。做法如下：

```
test=# CREATE TABLE t_data_2016 () INHERITS (t_data);
CREATE TABLE
test=# \d t_data_2016
                       Table "public.t_data_2016"
  Column |  Type   |                     Modifiers
---------+---------+-------------------------------------------------
 id      | integer | not null default
                     nextval('t_data_id_seq'::regclass)
 t       | date    |
 payload | text    |
Inherits: t_data
```

这个表被称作 t_data_2016，它从 t_data 继承而来。()表示没有为子表增加额外的列。如你所见，继承意味着来自父表的所有列都在子表中可用。还要注意 id 列将继承来自父

表的序列，因此所有的孩子能共享完全相同的编号。

让我们创建更多的表：

```
test=# CREATE TABLE t_data_2015 () INHERITS (t_data);
CREATE TABLE
test=# CREATE TABLE t data 2014 () INHERITS (t data);
CREATE TABLE
```

目前，所有的表都是相同的并且只是从父表继承。不过，其实可以做到更多，子表实际上可以有比其父辈更多的列。增加更多域很容易：

```
test=# CREATE TABLE t data 2013 (special text) INHERITS (t data);
CREATE TABLE
```

在这种情况下，增加了一个特殊列。它对父表没有影响，只是丰富了孩子表并且让它们能保存更多数据。

在创建了一些表后，可以增加一行：

```
test=# INSERT INTO t_data_2015 (t, payload)
   VALUES ('2015-05-04', 'some data');
INSERT 0 1
```

现在重头戏来了，父表可以用来查找子表中所有的数据：

```
test=# SELECT * FROM t_data;
 id |      t      |  payload
----+------------+-----------
 1  | 2015-05-04 | some data
(1 row)
```

查询父表允许用户以一种简单且有效的方式得到父表之下所有子表的访问。

要理解 PostgreSQL 如何做分区，有必要看看其计划：

```
test=# EXPLAIN SELECT * FROM t data;
                              QUERY PLAN
-----------------------------------------------------------------
 Append (cost=0.00..84.10 rows=4411 width=40)
   -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
   -> Seq Scan on t data 2016
         (cost=0.00..22.00 rows=1200 width=40)
   -> Seq Scan on t data 2015
         (cost=0.00..22.00 rows=1200 width=40)
   -> Seq Scan on t_data_2014
         (cost=0.00..22.00 rows=1200 width=40)
```

```
   -> Seq Scan on t_data_2013
        (cost=0.00..18.10 rows=810 width=40)
(6 rows)
```

实际上，该处理非常简单。PostgreSQL 简单地统一所有的表并且向我们展示所查看的分区及其下层分区中所有表的全部内容。注意所有的表都是独立的，它们只是通过系统目录在逻辑上被连接在一起。

6.5.2　应用表约束

如果应用过滤条件会发生什么？

```
test=# EXPLAIN SELECT * FROM t_data WHERE t = '2016-01-04';
                            QUERY PLAN
-----------------------------------------------------------------
 Append (cost=0.00..95.12 rows=23 width=40)
   -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
        Filter: (t = '2016-01-04'::date)
   -> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
        Filter: (t = '2016-01-04'::date)
   -> Seq Scan on t_data_2015 (cost=0.00..25.00 rows=6 width=40)
        Filter: (t = '2016-01-04'::date)
   -> Seq Scan on t_data_2014 (cost=0.00..25.00 rows=6 width=40)
        Filter: (t = '2016-01-04'::date)
   -> Seq Scan on t_data_2013 (cost=0.00..20.12 rows=4 width=40)
        Filter: (t = '2016-01-04'::date)
(11 rows)
```

PostgreSQL 把过滤条件应用在该结构中的所有分区上。它不知道表名与表的内容有某种相关性。对于数据库，名称只是名称并且与要查找的东西没有关系。当然，这很容易理解，因为没有数学证明可以用来做别的事情。

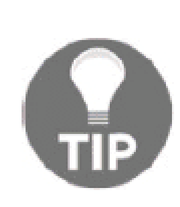
现在的问题是，我们怎么才能让数据库知道 2016 表只包含 2016 的数据，2015 表只包含 2015 的数据等？表约束就是这个用途的。它们会告诉 PostgreSQL 那些表的内容，因此让规划器做出比以前更聪明的决定。这种特性被称为约束排除，它能在很多情况下极大地加速查询。

下面的列表展示了如何创建表约束：

```
test=# ALTER TABLE t_data_2013
   ADD CHECK (t < '2014-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2014
```

```
   ADD CHECK (t >= '2014-01-01' AND t < '2015-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2015
   ADD CHECK (t >= '2015-01-01' AND t < '2016-01-01');
ALTER TABLE
test=# ALTER TABLE t_data_2016
   ADD CHECK (t >= '2016-01-01' AND t < '2017-01-01');
ALTER TABLE
```

每个表都可以增加一个 CHECK 约束。



PostgreSQL 只会在那些表中的所有数据都完全正确且每一行都满足约束时创建约束。与 MySQL 不同，PostgreSQL 中的约束在任何环境下都会被严肃对待和遵循。

在 PostgreSQL 中，那些约束可以重叠——这并未被禁止并且可以在一些情况下有意义。不过，创建非重叠的约束一般来说会更好，因为 PostgreSQL 可以剪枝更多表。

下面是增加了那些表约束后的效果：

```
test=# EXPLAIN SELECT *
   FROM        t_data
   WHERE       t = '2016-01-04';
                          QUERY PLAN
-----------------------------------------------------------------
 Append (cost=0.00..25.00 rows=7 width=40)
   -> Seq Scan on t_data (cost=0.00..0.00 rows=1 width=40)
        Filter: (t = '2016-01-04'::date)
   -> Seq Scan on t_data_2016 (cost=0.00..25.00 rows=6 width=40)
        Filter: (t = '2016-01-04'::date)
(5 rows)
```

规划器能够从查询中移除很多表，并且只保留那些可能会包含数据的表。查询可以从一个更短、更有效的计划中大大受益。尤其是如果那些表确实很大时，移除它们可以非常可观地提高速度。

6.5.3　修改继承的结构

有时数据结构必须被修改，“ALTER TABLE”子句就是做这个的。问题是怎么修改被分区的表？

基本上所有要做的事情就是处理父表并且增加或者移除列。PostgreSQL 自动地把那

些更改传播到子表并且确保那些更改被应用于所有的关系：

```
test=# ALTER TABLE t_data ADD COLUMN x int;
ALTER TABLE
test=# \d t_data_2016
                      Table "public.t_data_2016"
  Column |   Type  |                     Modifiers
---------+---------+--------------------------------------------------
 id      | integer | not null default
                     nextval('t data id seq'::regclass)
 t       | date    |
 payload | text    |
 x       | integer |
Check constraints:
   "t data 2016 t check"
   CHECK (t >= '2016-01-01'::date AND t < '2017-01-01'::date)
Inherits: t_data
```

如你所见，列被增加到父表并且被自动地增加到子表上。

注意这种方式对列之类的东西都有效，但索引则完全是另外一码事。在继承结构中，每一个表都必须被单独索引。如果对父表增加一个索引，它只会出现在父表上——而不会被部署在那些子表上。在所有那些表中索引所有那些列是用户的任务，PostgreSQL 不会帮用户做那些决定。当然，这可以被视作一种特性或者一种限制。从好的方面看，我们可以说 PostgreSQL 给了用户所有灵活性来单独索引表，可能效率因此更好。但是，人们可能也有争议，逐个部署所有这些索引的工作量太大。

6.5.4　在分区结构中移进和移出表

假定有一个继承结构，数据被按照日期分区并且想要对最终用户提供最近几年的数据。在某一时刻，用户可能想要把一些数据移出用户的视线而不实际触碰到数据。用户还可以把数据放入某种归档等。

PostgreSQL 提供了一种简单的方式来实现这一需求。首先可以创建一个新的父表：

```
test=# CREATE TABLE t history (LIKE t data);
CREATE TABLE
```

LIKE 关键词允许用户创建一个和 t_data 表具有完全相同布局的表。在忘记了 t_data 表实际有哪些列的情况下，这会非常方便，因为它减少了很多工作量。这种方式还可以包括索引、约束和默认值。

然后可以从旧的父表移除表并且把它们放在新的父表之下。

做法如下：

```
test=# ALTER TABLE t_data_2013 NO INHERIT t_data;
ALTER TABLE
test=# ALTER TABLE t_data_2013 INHERIT t_history;
ALTER TABLE
```

当然可以把整个处理放在一个事务中以确保操作的原子性。

6.5.5　清理数据

分区表的一个优点是可以快速清理数据。假定我们想要删除一整年的数据，如果数据已经被相应地分区好，用一个简单的 DROP TABLE 子句就能完成这一任务：

```
test=# DROP TABLE t_data_2014;
DROP TABLE
```

如你所见，删除一个子表很容易。但是删除父表会怎样呢？由于有依赖对象存在，因此 PostgreSQL 自然会报错来确保不会发生意外：

```
test=# DROP TABLE t data;
ERROR: cannot drop table t data because other objects depend on it
DETAIL: default for table t data 2013 column id depends on
   sequence t data id seq
table t_data_2016 depends on table t_data
table t_data_2015 depends on table t_data
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

“DROP TABLE”子句会警告有依赖对象存在且拒绝删除的那些表。强制 PostgreSQL 把那些对象和父表一起实际地删除，需要 CASCADE 子句：

```
test=# DROP TABLE t_data CASCADE;
NOTICE: drop cascades to 3 other objects
DETAIL: drop cascades to default for table t data 2013 column id
drop cascades to table t data 2016
drop cascades to table t data 2015
DROP TABLE
```

6.6　为好的查询性能调整参数

编写好的查询是达到好性能的第一步。没有好的查询，用户很可能会被不好的性能折磨。因此，编写好的并且聪明的代码将让用户更有可能得到好的性能。一旦查询已经

从逻辑和语义的角度完成优化，好的内存设置将提供最终的加速。在本节中，读者将学到更多的内存能为我们做什么以及 PostgreSQL 能怎样利用它来让我们受益。

为了说明问题，笔者编了一个简单的例子：

```
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t test (name) SELECT 'hans'
   FROM generate series(1, 100000);
INSERT 0 100000
test=# INSERT INTO t_test (name) SELECT 'paul'
   FROM generate_series(1, 100000);
INSERT 0 100000
```

一百万个包含 hans 的行将被增加到表中，然后一百万个包含 paul 的行被载入。表中将总共有两百万个唯一的 ID，但是只有两种不同的名字。

现在让我们使用 PostgreSQL 的默认内存设置运行一个简单的查询：

```
test=# SELECT name, count(*) FROM t test GROUP BY 1;
 name | count
------+--------
 hans | 100000
 paul | 100000
(2 rows)
```

不出意外，有两行将被返回。但这里的重点不是结果，而是 PostgreSQL 在幕后做了什么：

```
test=# EXPLAIN ANALYZE SELECT name, count(*)
   FROM t_test
   GROUP BY 1;
                              QUERY PLAN
-----------------------------------------------------------------
 HashAggregate (cost=4082.00..4082.01 rows=1 width=13)
   (actual time=51.448..51.448 rows=2 loops=1)
               Group Key: name
   -> Seq Scan on t_test
        (cost=0.00..3082.00 rows=200000 width=5)
        (actual time=0.007..14.150 rows=200000 loops=1)
 Planning time: 0.032 ms
 Execution time: 51.471 ms
(5 rows)
```

PostgreSQL 发现分组的数目实际上非常小。因此，它创建了一个哈希表，并且为每个分组加入了一个哈希项，然后开始计数。由于分组数量低，哈希表其实很小并且 PostgreSQL 能够很快地通过为每个分组增加数字完成计数。

如果通过 id 而不是 name 分组会发生什么？分组的数量将会飙升：

```
test=# EXPLAIN ANALYZE SELECT id, count(*)
 FROM t test
 GROUP BY 1;
 QUERY PLAN
-----------------------------------------------------------------
 GroupAggregate  (cost=23428.64..26928.64 rows=200000 width=12)
 (actual time=97.128..154.205 rows=200000 loops=1)
 Group Key: id
 -> Sort (cost=23428.64..23928.64 rows=200000 width=4)
 (actual time=97.120..113.017 rows=200000 loops=1)
 Sort Key: id
 Sort Method: external sort Disk: 2736kB
 -> Seq Scan on t_test
 (cost=0.00..3082.00 rows=200000 width=4)
 (actual time=0.017..19.469 rows=200000 loops=1)
 Planning time: 0.128 ms
 Execution time: 160.589 ms
(8 rows)
```

PostgreSQL 发现分组数现在大了很多，于是很快地改变了它的策略。问题是一个包含如此多项的哈希表无法放入内存中：

```
test=# SHOW work mem ;
 work mem
----------
 4MB
(1 row)
```

work_mem 变量掌管着 GROUP BY 子句使用的哈希表尺寸。由于有太多项，PostgreSQL 不得不找到一种不需要将整个数据集保持在内存中的策略。解决方案是按照 ID 排序数据并且将其分组。一旦数据被排序好，PostgreSQL 就能在列表中下移，然后一个接一个地构造分组。如果第一类值被计数完，则部分结果已经被算出并且可以被发出。然后可以处理下一个分组。在下移时一旦有序列表中的值改变，它就绝不会再次出现，因此系统知道一个部分结果已经准备好。

为了加速该查询，可以临时（当然，也可以从全局设置）为 work_mem 变量设置一

个较高的值：

```
test=# SET work_mem TO '1 GB';
SET
```

现在计划将再次由一个快速且有效的哈希聚集唱主角：

```
test=# EXPLAIN ANALYZE SELECT id, count(*) FROM t_test GROUP BY 1;
 QUERY PLAN
-----------------------------------------------------------------
 HashAggregate (cost=4082.00..6082.00 rows=200000 width=12)
 (actual time=76.967..118.926 rows=200000 loops=1)
 Group Key: id
 -> Seq Scan on t test
 (cost=0.00..3082.00 rows=200000 width=4)
 (actual time=0.008..13.570 rows=200000 loops=1)
 Planning time: 0.073 ms
 Execution time: 126.456 ms
(5 rows)
```

PostgreSQL 知道（或者至少会假设）数据将适合于内存并且切换到较快的计划。如你所见，执行时间变得更低。该查询不会和“GROUP BY name”情况一样快，因为不得不计算更多的哈希值，但用户在绝大多数情况下都将从中受益。

6.6.1　加速排序

work_mem 变量不仅能加速分组，还能对排序等简单的操作产生非常好的影响。排序可以说是世界上每一个数据库系统都要掌握的核心机制。

下列查询展示了一个使用默认 4MB 设置的简单操作：

```
test=# SET work mem TO default;
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
 QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=24111.14..24611.14 rows=200000 width=9)
 (actual time=219.298..235.008 rows=200000 loops=1)
 Sort Key: name, id
 Sort Method: external sort Disk: 3712kB
 -> Seq Scan on t test
 (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.006..13.807 rows=200000 loops=1)
```

```
 Planning time: 0.064 ms
 Execution time: 241.375 ms
(6 rows)
```

PostgreSQL 需要 13.8 毫秒来读取数据，并且需要超过 200 毫秒来排序数据。由于可用的内存量低，必须使用临时文件来执行排序。external sort Disk 方法只需要少量的 RAM，但必须把中间数据发送到相对较慢的存储设备，这当然会导致可怜的吞吐。

增加 work_mem 变量的设置将让 PostgreSQL 使用更多内存排序：

```
test=# SET work mem TO '1 GB';
SET
test=# EXPLAIN ANALYZE SELECT * FROM t_test ORDER BY name, id;
 QUERY PLAN
-----------------------------------------------------------------
 Sort (cost=20691.64..21191.64 rows=200000 width=9)
 (actual time=36.481..47.899 rows=200000 loops=1)
 Sort Key: name, id
 Sort Method: quicksort Memory: 15520kB
 -> Seq Scan on t test
 (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.010..14.232 rows=200000 loops=1)
 Planning time: 0.037 ms
 Execution time: 55.520 ms
(6 rows)
```

由于现在有足够的内存，数据库将在内存中做所有的排序并且因此会极大地提高处理速度。现在排序只需要 33 毫秒，这比之前查询提升了 7 倍。更多的内存将导致更快的排序并且提高系统的速度。

到目前为止，读者已经看到了两种排序数据的机制：external sort Disk 和 quicksort Memory。除这两种机制之外，还有第三种算法，它是“top-N heapsort Memory”。它只能被用来提供前 N 行：

```
test=# EXPLAIN ANALYZE SELECT *
 FROM t_test
 ORDER BY name, id
 LIMIT 10;
 QUERY PLAN
-----------------------------------------------------------------
 Limit (cost=7403.93..7403.95 rows=10 width=9)
 (actual time=31.837..31.838 rows=10 loops=1)
 -> Sort (cost=7403.93..7903.93 rows=200000 width=9)
```

```
 (actual time=31.836..31.837 rows=10 loops=1)
 Sort Key: name, id
 Sort Method: top-N heapsort Memory: 25kB
 -> Seq Scan on t test
 (cost=0.00..3082.00 rows=200000 width=9)
 (actual time=0.011..13.645 rows=200000 loops=1)
 Planning time: 0.053 ms
 Execution time: 31.856 ms
(7 rows)
```

该算法快如闪电，整个查询将在比 30 毫秒略多的时间内完成。排序部分现在只需要 18 毫秒，这几乎和第一种情况中读取数据一样快。

注意，work_mem 变量是对每个操作分配的。理论上会发生一个查询需要多于一次的 work_mem 变量。它不是一个全局设置——它实际上是针对每个操作的。因此必须以一种小心的方式设置它。

读者应该记住一件事情，很多书籍声称在 OLTP 系统上将 work_mem 设置得过高可能会导致服务器内存不足。是的，如果 1000 个人同时排序 100MB 数据，就会导致内存失效。但是，磁盘能够处理这么大的数据量吗？笔者很怀疑。解决方案只能是：停止做愚蠢的事情。无论如何，并发地排序 100MB 数据 1000 次不应该是一个 OLTP 系统中应该发生的事情。应该考虑部署合适的索引、编写更好的查询或者简单地重新思考所面对的需求。在任何情况下，过于频繁地并发排序那么大量的数据都不是什么好主意——在那些事情造成应用停摆之前赶快停下来。

6.6.2　加速管理任务

还有更多操作不得不做某种排序或者某种类型的内存分配。CREATE INDEX 子句之类的管理操作不依赖于 work_mem 变量，而是使用 maintenance_work_mem 变量。用法如下：

```
test=# SET maintenance_work_mem TO '1 MB';
SET
test=# \timing
Timing is on.
test=# CREATE INDEX idx id ON t test (id);
CREATE INDEX
Time: 104.268 ms
```

如你所见，在两百万行上创建一个索引花了大约 100 毫秒，这个速度确实有点慢。因此，可以用 maintenance_work_mem 变量来加速排序，而“CREATE INDEX”子句本质上就依赖于排序：

```
test=# SET maintenance_work_mem TO '1 GB';
SET
test=# CREATE INDEX idx_id2 ON t_test (id);
CREATE INDEX
Time: 46.774 ms
```

现在的速度翻了一倍，正是因为排序被提升了那么多。

还有更多管理任务会从使用更多内存中受益，最突出的是 VACUUM 子句（清除索引）和 ALTER TABLE 子句。maintenance_work_mem 变量的规则与 work_mem 变量相同。设置是针对每个操作的，并且所需内存只在使用时才分配。

6.7 总　　结

在本章中讨论了许多查询优化。读者学到了有关优化器和多种内部优化的内容，例如常数折叠、视图内联、连接等。所有这些优化都有助于好的性能，并且能够可观地提高速度。

在介绍了优化之后，第 7 章将涉及存储过程。读者将看到 PostgreSQL 用于处理用户定义代码的选项。

第 7 章　编写存储过程

在第 6 章中，读者已经学到了很多有关优化器的内容以及系统中所进行的优化。本章将介绍存储过程以及如何有效和方便地使用它们。读者将学到存储过程的构成、可用的语言以及如何很好地提高它们的速度。除此之外，读者还将了解 PL/pgSQL 的一些更高级的特性。

本章将涵盖下列内容：

- 决定正确的语言。
- 存储过程如何被执行。
- PL/pgSQL 的高级特性。
- 打包扩展。
- 优化性能。
- 配置函数参数。

在本章结束时，读者将能够编写优秀且高效的过程。

7.1　理解存储过程语言

在存储过程方面，PostgreSQL 和其他数据库系统有相当显著的区别。大部分数据库引擎强制用户使用一种特定的编程语言来编写服务器端的代码。微软 SQL Server 提供了 Transact-SQL，而 Oracle 鼓励用户使用 PL/SQL。PostgreSQL 不强制用户使用一种特定的语言，而是允许用户决定自己最了解以及最喜欢的语言。

在历史上，PostgreSQL 如此灵活的原因实际上也非常有趣。很多年前，最有名的 PostgreSQL 开发者之一（Jan Wieck，在早期编写了无数的补丁）提出了用 TCL 作为服务器端编程语言的想法。但问题很简单——没有人想用 TCL 并且没有人想在数据库引擎里加上它。对于这个问题的解决方案就变成了让语言接口如此灵活的现状，基本上任何语言都可以很容易地与 PostgreSQL 整合起来。然后， CREATE LANGUAGE 子句便诞生了：

```
test=# h CREATE LANGUAGE
Command:     CREATE LANGUAGE
Description: define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
```

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ]
            [ VALIDATOR valfunction ]
```

现如今，很多不同的语言都可以被用来编写存储过程。早期加入 PostgreSQL 的这种灵活性确实取得了成功，因此用户可以从丰富的编程语言集合中进行选择。

PostgreSQL 到底是怎样处理语言的？如果读者看一看 CREATE LANGUAGE 子句的语法，那么就能看到几个关键词。

- HANDLER：这个函数实际上就是 PostgreSQL 和任何想要使用的外部语言之间的粘合剂。它负责将 PostgreSQL 数据结构映射到语言所需的结构并且帮助传递给代码。
- VALIDATOR：这是整个架构中的警察。如果这个部件可用，它负责将语法错误递送给最终用户。很多语言都能够在真正执行代码前先解析它。PostgreSQL 可以使用这一点并且在创建函数时告诉用户它是否正确。不幸的是，并非所有的语言都能这样做，因此在某些情况下，问题只会在运行时出现。
- INLINE：如果它存在，PostgreSQL 将能够在这个函数内运行匿名代码块。
- 存储过程剖析

在实际深入一种特定语言之前，笔者想先谈谈存储过程的典型构造。为了演示，笔者已经编写好了一个函数，它只是将两个数加起来：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
'
  SELECT $1 + $2;
' LANGUAGE 'sql';
CREATE FUNCTION
```

可以看到的第一点是，该过程采用 SQL 编写。PostgreSQL 必须知道我们使用哪种语言，因此我们必须在定义中指定这一点。注意该函数的代码被作为一个字符串（'）传递给 PostgreSQL。这是值得注意的地方，因为这样做会把函数当作一个黑盒交给执行机制。在其他数据库引擎中，函数的代码不是字符串，而是直接附在语句中。这种简单的抽象层就是 PostgreSQL 函数管理器的力量来源。

在该字符串中，用户基本上可以使用所有的编程语言。在笔者的例子中，笔者只是简单地将传递给函数的两个数加起来。对于这个例子，用到了两个整数变量。这里的重点是 PostgreSQL 提供了函数重载。换句话说，mysum(int, int)和 mysum(int8, int8)是不同的函数，PostgreSQL 将这些东西视作两个不同的函数。虽然函数重载是一种很好的特

性，但是，如果用户的函数参数列表碰巧会随时间变化，那么就要非常小心不要部署过多函数。一定要确保删除不再需要的函数。



CREATE OR REPLACE FUNCTION 子句将不会更改参数列表。因此，只有签名没有改变时才能使用它。它将报出错误或者简单地部署一个新的函数。

让我们运行这个函数：

```
test=# SELECT mysum(10, 20);
 mysum
-------
    30
(1 row)
```

结果确实没什么稀奇的。

1. 引入美元符号引用

将代码作为字符串传递给 PostgreSQL 非常灵活。但是，使用单引号可能会是一个问题。在很多编程语言中，单引号会很频繁地出现。为了能够使用引号，人们不得不在将字符串传递给 PostgreSQL 时对它们转义。很多年来，这都是标准程序。幸运的是，旧时代已经过去，现在有新方法可以传递代码给 PostgreSQL：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$$
    SELECT $1 + $2;
$$ LANGUAGE 'sql';
CREATE FUNCTION
```

字符串引用问题的解决方案被称作美元引用。除了使用引号开始和结束字符串之外，用户还可以使用$$。当前，笔者只发现两种语言为$$赋予了含义。在 Perl 以及 bash 脚本中，$$表示进程 ID。为了克服这个小障碍，用户可以使用“$几乎所有东西$”来开始和结束字符串。下面的例子演示了这种用法：

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$__$
     SELECT $1 + $2;
$__$ LANGUAGE 'sql';
CREATE FUNCTION
```

所有这种灵活性允许用户真正一劳永逸地克服引用的问题。只要开始字符串和结束字符串匹配，就不会有任何问题。

2. 使用匿名代码块

到目前为止，读者已经学到了编写最简单的存储过程以及执行代码。但是代码执行中不仅仅只有典型的存储过程。除典型的存储过程之外，PostgreSQL 还允许使用匿名代码块。其思想是运行只需要一次的代码。这种代码执行在处理管理任务时尤其有用。匿名代码块没有参数，并且不会被持久存储在数据库中，因为不管怎样它们都不会有名字。

这里有一个简单的例子：

```
test=# DO
$$
    BEGIN
            RAISE NOTICE 'current time: %', now();
    END;
$$ LANGUAGE 'plpgsql';
NOTICE: current time: 2016-12-12 15:25:50.678922+01
CONTEXT: PL/pgSQL function inline code block line 3 at RAISE
DO
```

在这个例子中，代码只是发出一个消息并且退出。同样，代码块必须知道它使用哪种语言。字符串也再次通过使用简单的美元引用传递给 PostgreSQL。

3. 使用函数和事务

如你所知，PostgreSQL 暴露在用户空间的所有东西都是一个事务。当然，编写存储过程时也是一样。过程总是所在的事务的一部分，它不是自治的——它就像一个操作符或者任何其他操作。

这里是一个例子：

```
test=# SELECT now(), mysum(id, id) FROM generate_series(1, 3) AS id;
              now              | mysum
-------------------------------+-------
 2016-12-12 15:54:32.287027+01 | 2
 2016-12-12 15:54:32.287027+01 | 4
 2016-12-12 15:54:32.287027+01 | 6
(3 rows)
```

所有 3 个函数调用都发生在同一个事务中。理解这一点很重要，因为它意味着在一个函数内无法做太多事务性的流程控制。假定第二个函数调用提交，这种情况下会发生

什么？这样是无法工作的。

但是 Oracle 有一种机制允许自治事务。其想法是如果一个事务回滚，某些部分可能仍被需要并且应该被保留。经典的例子如下：

- 开始一个函数查找秘密数据。
- 在文档中增加一个日志行说明有人已经修改了这个重要的秘密数据。
- 提交日志行但是回滚更改。
- 仍需要知道某人尝试过更改数据。

要解决这样的问题，可以使用自治事务。其思想是在主事务内独立地提交一个事务。在这种情况下，日志表中的项将留下来而更改将被回滚。

截至 PostgreSQL 9.6，自治事务还没有出现。但是，笔者已经见过一些实现这一特性的补丁到处流传。至于这些特性何时会被放到核心中，就让我们拭目以待吧。

为了让读者对这些特性有些印象，这里有一个基于第一批补丁的代码片段：

```
...
AS $$
DECLARE
 PRAGMA AUTONOMOUS TRANSACTION;
BEGIN
 FOR i IN 0..9 LOOP
   START TRANSACTION;
   INSERT INTO test1 VALUES (i);
   IF i % 2 = 0 THEN
       COMMIT;
   ELSE
       ROLLBACK;
   END IF;
 END LOOP;

 RETURN 42;
END;
$$;
...
```

这个例子中的重点是，我们可以临时决定提交或者回滚自治事务。

7.2　理解各种存储过程语言

正如本章开始时所述，PostgreSQL 让用户能够使用多种语言编写存储过程。PostgreSQL

核心包含了下列选项：

- SQL
- PL/pgSQL
- PL/Perl 和 PL/PerlU
- PL/Python
- PL/Tcl 和 PL/TclU

SQL 是编写存储过程最显而易见的选择，应该尽可能使用它，因为它为优化器提供了最大的自由度。不过，如果用户想要编写更加复杂的代码，PL/pgSQL 可能是一种选择。它提供了流程控制和更多的特性。本章将展示 PL/pgSQL 的一些更高级和比较少见的特性（本章并不想成为 PL/pgSQL 的一份完整教程）。

然后，核心还包括运行以 Perl 编写的存储过程的代码。说白了，这里的逻辑是相同的。代码将被作为一个字符串传递并且由 Perl 执行。PostgreSQL 并不会 Perl 语言——它仅仅是将代码转交给外部编程语言而已。

也许读者已经注意到了 Perl 和 TCL 都有两种形式：可信的（PL/Perl 和 PL/TCL）以及不可信的（PL/PerlU 和 PL/TCLU）。可信的和不可信的语言之间的区别实际上很重要。在 PostgreSQL 中，一种语言会被直接载入数据库连接中。因此，该语言能够做很多很不好的事情。为了除去安全性问题，可信语言的概念被发明出来。其思想是可信语言被限制在该语言非常核心的部分中，它不能做下面的事情：

- 包括库。
- 打开网络套接字。
- 执行任何类型的系统调用（打开文件等）。

Perl 提供了感染模式，它被用来实现 PostgreSQL 中的这种特性。Perl 将自动把自身限制在可信模式，在将要违背安全性时报错。在不可信模式中，所有的事情都可以发生，因此只有超级用户被允许运行不可信代码。

如果想要运行可信的以及不可信的代码，必须激活两种语言：plperl 和 plperlu（相应的还有 pltcl 和 pltclu）。

Python 当前只能作为一种不可信语言使用，因此，在涉及安全性时管理员必须非常小心，因为运行在不可信模式的存储过程可能绕过 PostgreSQL 实施的所有安全性机制。不管怎样，只要记住一点就好，Python 是作为数据库连接的一部分运行并且绝不对安全性负责。

7.2.1　引入 PL/pgSQL

让我们从最令人期待的话题开始，笔者相信读者将会热衷于多了解一些有关于它的

内容。

本节将介绍 PL/pgSQL 的一些更高级的特性，它们对于编写正确且高效的代码非常重要。注意这并非是针对新手的编程或者 PL/pgSQL 介绍。

1. 处理引用

数据库编程中最重要的事情之一是引用。如果没有使用正确的引用，肯定会受到 SQL 注入以及开放的不可接受的安全漏洞困扰。

什么是 SQL 注入？考虑下面的例子：

```
CREATE FUNCTION broken (text)
RETURNS void AS
$$
        DECLARE
                v_sql text;
        BEGIN
                v_sql := 'SELECT schemaname
                                FROM pg tables
                                WHERE tablename = ''' || $1 || '''';
                RAISE NOTICE 'v sql: %', v sql;
                RETURN;
        END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，SQL 代码被简单地粘贴在一起，根本不担心安全性。这里所做的一切就是使用“||”操作符串接字符串。如果人们运行正常的查询，这会工作得很好：

```
SELECT broken('t_test');
```

但是，我们必须时刻防备坏人。考虑下面的例子：

```
SELECT broken('''; DROP TABLE t_test; ');
```

用这个参数运行该函数会给我们带来一个吸引人的小问题：

```
NOTICE: v_sql: SELECT schemaname FROM pg_tables
   WHERE tablename = ''; DROP TABLE t_test; '
CONTEXT: PL/pgSQL function broken(text) line 6 at RAISE
 broken
--------


(1 row)
```

在只想查找时却删除了一个表，这当然不是我们想要做的事情。基于传递给语句的

参数来建立应用的安全性绝对不可接受。

为了避免 SQL 注入，PostgreSQL 提供了多种函数，应该一直使用它们来确保安全性不受损伤：

```
test=# SELECT quote_literal(E'o''reilly'),
                 quote_ident(E'o''reilly');
 quote_literal | quote_ident
---------------+-------------
 'o''reilly'   | "o'reilly"
(1 row)
```

quote_literal 函数转义字符串的方式不会再发生不好的事。它在字符串周围加上所有的引号，转义字符串内有问题的字符。因此，就不再需要手工开始和结束字符串。

这里展示的第二个函数是 quote_ident。它可以被用来正确地引用对象名称。注意它会使用双引号，这正是处理表名和类似东西所需要的：

```
test=# CREATE TABLE "Some stupid name" ("ID" int);
CREATE TABLE
test=# d "Some stupid name"
Table "public.Some stupid name"
 Column |   Type   | Modifiers
--------+---------+-----------
 ID     | integer |
```

通常，PostgreSQL 中所有的表名都是小写。但是，如果使用了双引号，对象名称可以包含大写字母。一般来说，这种技巧并不是什么好主意，因为这种情况下用户不得不始终使用双引号，这有点不方便。

在引用的基本介绍之后，有必要看看 NULL 值是如何被处理的：

```
test=# SELECT quote_literal(NULL);
 quote_literal
---------------

(1 row)
```

如果在一个 NULL 值上调用 quote_literal 函数，它将简单地返回 NULL。这种情况下也没有必要关注引用。

PostgreSQL 甚至提供了更多函数来明确地应对 NULL 值：

```
test=# SELECT quote_nullable(123),
                  quote_nullable(NULL);
 quote_nullable | quote_nullable
```

```
----------------+----------------
 '123'          | NULL
(1 row)
```

不仅可以引用字符串和对象名称，还可以使用 PL/pgSQL 自带的方法来格式化和准备整个查询。这里的妙处是用户可以使用 format 函数为语句增加参数。下面是用法：

```
CREATE FUNCTION simple format()
RETURNS text AS
$$
   DECLARE
          v_string       text;
          v_result       text;
   BEGIN
          v string := format('SELECT schemaname
                                || '' .'' || tablename
                         FROM pg tables
                         WHERE %I = $1
                         AND %I = $2', 'schemaname', 'tablename');
          EXECUTE v_string USING 'public', 't_test' INTO v_result;
          RAISE NOTICE 'result: %', v_result;
          RETURN v_string;
   END;
$$ LANGUAGE 'plpgsql';
```

域的名称被传递给 format 函数。最后，EXECUTE 语句的 USING 子句把参数增加到查询中，然后查询被执行。再一次，这里的好处是不会发生 SQL 注入。

下面是执行后的效果：

```
test=# SELECT simple format ();
NOTICE: result: public .t test
CONTEXT: PL/pgSQL function simple format() line 12 at RAISE
                        simple_format
---------------------------------------------------------------
 SELECT schemaname                                            +
                                          || ' .' || tablename +
                                   FROM  pg tables            +
                                   WHERE schemaname = $1      +
                                   AND tablename = $2
(1 row)
```

如你所见，调试信息正确地显示了包括方案的表名并且正确地返回了查询语句。

2. 管理作用域

在大体上处理了引用和基本安全性（SQL 注入）之后，笔者想要把读者的焦点转移到另一个重要的主题：作用域。

正如笔者所知道的最流行的编程语言一样，PL/pgSQL 依赖于上下文环境使用变量。变量在一个函数的 DECLARE 语句中定义。不过，PL/pgSQL 允许用户嵌套 DECLARE 语句：

```
CREATE FUNCTION scope test ()
RETURNS int AS
$$
   DECLARE
          i int := 0;
   BEGIN
          RAISE NOTICE 'i1: %', i;
          DECLARE
                 i int;
          BEGIN
                 RAISE NOTICE 'i2: %', i;
          END;
          RETURN i;
   END;
$$ LANGUAGE 'plpgsql';
```

DECLARE 语句定义了一个变量 i 并且为它进行了赋值，然后 i 被显示，输出当然将会是 0。然后第二个 DECLARE 语句开始。它包含 i 的一个额外的化身，但没有被赋值。因此，该值将会是 NULL。注意现在 PostgreSQL 显示内层的 i。结果如下：

```
test=# SELECT scope test();
NOTICE: i1: 0
CONTEXT: PL/pgSQL function scope test() line 5 at RAISE
NOTICE: i2: <NULL>
CONTEXT: PL/pgSQL function scope_test() line 10 at RAISE
 scope_test
------------
          0
(1 row)
```

不出所料，调试信息将显示 0 和 NULL。



PostgreSQL 允许用户运用各种各样的技巧。不过，强烈推荐保持代码的简单和易读。

3. 理解高级错误处理

在每种编程语言、每个程序以及每个模块中，错误处理都是一件重要的事情。每件事情都偶尔会出错，因此正确和专业地处理错误是至关重要的。在 PL/pgSQL 中，用户可以使用 EXCEPTION 块来处理错误。其思想是，如果 BEGIN 块做错了什么事情，EXCEPTION 块将注意到并且正确地处理问题。就像很多其他语言（如 Java）一样，用户可以分别对不同类型的错误做出反应并且捕获它们。

在下面的例子中，代码将会出现一个除零问题。我们的目标就是捕捉到这个错误并且相应地做出反应：

```
CREATE FUNCTION error_test1(int, int)
RETURNS int AS
$$
   BEGIN
      RAISE NOTICE 'debug message: % / %', $1, $2;
         BEGIN
            RETURN $1 / $2;
         EXCEPTION
            WHEN division_by_zero THEN
               RAISE NOTICE 'division by zero detected: %',
                         sqlerrm;
            WHEN others THEN
               RAISE NOTICE 'some other error: %',
                         sqlerrm;
            END;

            RAISE NOTICE 'all errors handled';
            RETURN 0;
   END;
$$ LANGUAGE 'plpgsql';
```

BEGIN 块能够明确地抛出错误。不过，EXCEPTION 块会捕捉到我们所要考虑的错误并且还会注意可能意外出现的所有其他问题。

技术上，这或多或少与保存点相同，因此错误不会导致整个事务完全失败。只有导致该错误的块将会遭受微型的回滚。

通过检查 sqlerrm 变量，用户还可以直接访问错误消息本身。让我们运行代码：

```
test=# SELECT error_test1(9, 0);
NOTICE: debug message: 9 / 0
CONTEXT: PL/pgSQL function error_test1(integer,integer) line 3 at RAISE
```

```
NOTICE: division by zero detected: division by zero
CONTEXT: PL/pgSQL function error_test1(integer,integer) line 9 at RAISE
NOTICE: all errors handled
CONTEXT: PL/pgSQL function error test1(integer,integer) line 14 at RAISE
 error test1
-------------
           0
(1 row)
```

PostgreSQL 在 EXCEPTION 块中捕捉该异常并且显示消息。PostgreSQL 已经足够友好地告诉我们错误发生在哪一行以及出现问题时怎样能更容易地调试和修复代码。

在某些情况下，还有必要抛出用户自己的异常。如你所料，这很容易做到：

```
RAISE unique_violation
   USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

除了已经看到的内容之外，PostgreSQL 提供了很多预定义的错误代码和异常。下面的页面包含了这些错误消息的完整列表：https://www.postgresql.org/docs/9.6/static/errcodes-appendix.html。

4. 使用 GET DIAGNOSTICS

读者中很多曾用过 Oracle 的人可能很熟悉 GET DIAGNOSTICS 子句。GET DIAGNOSTICS 子句的思想是允许用户看到系统中在进行着什么。虽然这种语法对于习惯了现代代码的人来说可能有些奇怪，它仍然是一种可以用来让应用变得更好的有价值的工具。

在笔者看来，GET DIAGNOSTICS 子句可用于以下两种主要任务：

- 检查行计数。
- 取得上下文信息并且得到回溯。

检查行计数绝对是在日常编程中需要做的一件事情。而如果想要调试应用，提取上下文信息将会很有用。

下面的例子展示了如何在代码中使用 GET DIAGNOSTICS 子句：

```
CREATE FUNCTION get diag()
RETURNS int AS
$$
   DECLARE
          rc int;

          _sqlstate text;
```

```
          _message text;
          _context text;

   BEGIN
          EXECUTE 'SELECT * FROM generate series(1, 10)';
          GET DIAGNOSTICS rc = ROW COUNT;
          RAISE NOTICE 'row count: %', rc;

          SELECT rc / 0;
   EXCEPTION
          WHEN OTHERS THEN

                GET STACKED DIAGNOSTICS
                        sqlstate = returned sqlstate,
                        message = message text,
                       _context = pg_exception_context;
                RAISE NOTICE 'sqlstate: %, message: %,
                       context: [%]',
                        sqlstate,  message,
                       replace( context, E'n', ' <- ');
          RETURN rc;
   END;
$$ LANGUAGE 'plpgsql';
```

定义那些变量之后的第一件事情是执行一个 SQL 语句，并且向 GET DIAGNOSTICS 子句要求行计数，接着行计数会被显示在调试信息中。

然后该函数强制 PL/pgSQL 报错。一旦报错，就可以使用 GET DIAGNOSTICS 子句从服务器提取信息显示。

下面是执行的效果：

```
test=# SELECT get_diag();
NOTICE: row count: 10
CONTEXT: PL/pgSQL function get diag() line 12 at RAISE
NOTICE: sqlstate: 22012,
            message: division by zero,
            context: [SQL statement "SELECT rc / 0"
                   <- PL/pgSQL function get_diag() line 14 at
                          SQL statement]
CONTEXT: PL/pgSQL function get_diag() line 22 at RAISE
 get diag
----------
```

```
        10
(1 row)
```

如你所见，GET DIAGNOSTICS 子句对系统中正在进行什么给出了非常详细的信息。

5. 在块中使用游标取数据

如果用户执行 SQL，数据库将计算结果并且把结果发给用户的应用。

一旦整个结果集被发送到客户端，应用就可以继续做自己的工作。但问题是：如果结果集大到无法再放入内存中，会发生什么？如果数据库返回 100 亿行会怎样？客户端应用通常不能一次处理这么多的数据并且它也不应该这样做。这类问题的解决方案是游标。游标的思想是只在需要数据时（调用 FETCH 时）才产生数据。因此，在数据实际被数据库生成时应用可能已经开始消耗数据。此外，执行操作所需的内存也低很多。

对于 PL/pgSQL，游标也扮演了重要的角色。只要对一个结果集进行循环，PostgreSQL 在内部就将自动使用一个游标。其优点在于应用的内存消耗将被极大地降低并且很少会出现由于处理大量数据导致的内存不足现象。有多种方法使用游标，下面是最简单的一种：

```
CREATE FUNCTION c (int)
RETURNS setof text AS
$$
   DECLARE
          v_rec          record;
   BEGIN
          FOR v_rec IN SELECT tablename FROM pg_tables LIMIT $1
          LOOP
                 RETURN NEXT v_rec.tablename;
          END LOOP;

          RETURN;
   END;
$$ LANGUAGE 'plpgsql';
```

这段代码有两个有意思的地方。首先，它是一个集合返回函数（SRF）。它产生一整列而不只是单个行。实现这一点的方式是使用 setof 变量而不只是数据类型。RETURN NEXT 子句将构建结果集，直到我们到达结尾处。RETURN 子句会告诉 PostgreSQL 我们想要离开函数并且结果已经形成。

第二个重要的问题是在游标上循环将自动创建一个内部游标。换句话说，没有必要担心可能会出现内存不足。PostgreSQL 将以一种尽可能快产生前 10%数据（由 cursor_tuple_fraction 变量定义）的方式来优化查询。

下面是该查询将返回的结果：

```
test=# SELECT * FROM c(3);
c
--------------
 t test
 pg statistic
 pg type
(3 rows)
```

在这个例子中，结果将仅仅是一个随机表组成的列表。如果在读者的环境中产生的结果不同，那也是意料之中的。

在笔者看来，我们已经看到的是在 PL/pgSQL 中使用隐式游标的最频繁和最常见的方式。下面的例子展示了一种更古老的机制，很多以前使用 Oracle 的人可能都知道：

```
CREATE FUNCTION d (int)
RETURNS setof text AS
$$
   DECLARE
          v cur refcursor;
          v data text;
   BEGIN
          OPEN v_cur FOR SELECT tablename FROM pg_tables LIMIT $1;

          WHILE true
          LOOP
                 FETCH v cur INTO v data;
                 IF FOUND THEN
                        RETURN NEXT v data;
                 ELSE
                        RETURN;
                 END IF;
          END LOOP;
   END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，游标被显式地声明和打开。在内部，循环数据接着被显式取得并且返回给调用者。基本上，该查询做的是和之前完全相同的事情——这只是开发者实际更喜欢哪种语法的问题。

读者们是否仍然觉得对游标了解得还不够多？确实有，下面是第三种做同样事情的方法：

```
CREATE FUNCTION e (int)
RETURNS setof text AS
$$
   DECLARE
          v_cur CURSOR (param1 int) FOR
                 SELECT tablename FROM pg_tables LIMIT param1;
          v_data text;
   BEGIN
          OPEN v_cur ($1);

          WHILE true
          LOOP
                 FETCH v_cur INTO v_data;
                 IF FOUND THEN
                        RETURN NEXT v_data;
                 ELSE
                        RETURN;
                 END IF;
          END LOOP;
   END;
$$ LANGUAGE 'plpgsql';
```

在这种情况下，游标得到一个整数参数，它直接来自于函数调用（$1）。

有时，存储过程并未用尽一个游标，而是将它返回以便后续使用。在这种情况下，可以使用 refcursor 作为返回值：

```
CREATE OR REPLACE FUNCTION cursor_test(c refcursor)
RETURNS refcursor AS $$
BEGIN
OPEN c FOR
SELECT * FROM generate_series(1, 10) AS id;
RETURN c;
END;
$$ LANGUAGE plpgsql;
```

这里的逻辑很简单，游标的名字被传递给函数，然后该游标被打开并且返回。这里的好处是游标背后的查询可以被即时创建并且被动态编译。

应用可以就像在任何其他应用中那样从游标中取数据。下面是用法，注意只有使用了一个事务块时才能如此：

```
test=# BEGIN;
BEGIN
```

```
test=# SELECT cursor_test('mytest');
 cursor_test
-------------
 mytest
(1 row)

test=# FETCH NEXT FROM mytest;
 id
----
  1
(1 row)

test=# FETCH NEXT FROM mytest;
 id
----
  2
(1 row)
```

实际上，在本节中学到的是游标会根据消耗量来产生数据，对于大部分应用来说也确实是这样。但是，笔者在这个例子中加入了一个小圈套，只要使用 SRF，整个结果就不得不被物化。它不会被即时创建，而是一次性创建。原因在于，SQL 必须能够重新扫描一个关系，在普通表的情况下可能很容易做到。但是，函数的情况就不同了。因此，SRF 总是会被计算并且物化，这就让这个例子中的游标变得完全无用。换句话说，在编写函数时要小心——在某些情况下，危险就隐藏在精巧的细节中。

6. 利用组合类型

在大部分其他数据库系统中，存储过程只用于简单数据类型，例如 integer、numeric、varchar 等。不过，PostgreSQL 完全不同，用户基本上可以使用所有可用的数据类型，这包括基本类型以及组合类型和自定义类型。就所涉及的数据类型而言，可以说根本没有限制。为了发挥出 PostgreSQL 的全部实力，组合类型非常重要并且经常会被互联网上的各种扩展使用。

下面的例子展示了组合类型如何被传递给函数以及它在内部被如何使用。最后，组合类型将被返回：

```
CREATE TYPE my_cool_type AS (s text, t text);

CREATE FUNCTION f(my cool type)
RETURNS my cool type AS
$$
```

```
   DECLARE
          v_row my_cool_type;
   BEGIN
          RAISE NOTICE 'schema: (%) / table: (%)', $1.s, $1.t;
          SELECT schemaname, tablename INTO v row
                FROM   pg tables
                WHERE  tablename = trim($1.t)
                       AND schemaname = trim($1.s)
                LIMIT 1 ;
          RETURN v_row;
   END;
$$ LANGUAGE 'plpgsql';
```

这里的主要问题是用户可以简单地使用$1.field_name 来访问组合类型。返回类型也不难，用户只需要即时组装组合类型变量并且像其他数据类型那样返回即可。用户甚至可以轻易地使用数组或者更加复杂的结构。

下面的示例展示了 PostgreSQL 将返回的结果：

```
test=# SELECT (f).s, (f).t
            FROM f ('("public", "t test")'::my cool type);
NOTICE: schema: (public) / table: ( t_test)
CONTEXT: PL/pgSQL function f(my_cool_type) line 5 at RAISE
    s    |    t
--------+--------
 public | t test
(1 row)
```

7. 用 PL/pgSQL 编写触发器

如果用户想要对数据库中发生的特定事件做出反应，服务器端代码特别受欢迎。触发器允许用户在表上发生 INSERT、UPDATE、DELETE 或者 TRUNCATE 子句时调用一个函数。然后被触发器调用的函数修改表中发生改变的数据或者简单地执行某个需要的操作。

在 PostgreSQL 中，触发器这些年来已经变得更加强大并且提供了丰富的特性集合：

```
test=# h CREATE TRIGGER
Command:     CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }
            { event [ OR ... ] }
```

```
   ON table_name
   [ FROM referenced_table_name ]
   [ NOT DEFERRABLE | [ DEFERRABLE ]
    [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
   [ FOR [ EACH ] { ROW | STATEMENT } ]
   [ WHEN ( condition ) ]
   EXECUTE PROCEDURE function_name ( arguments )
```

这里，事件可以是下列之一：

```
INSERT
   UPDATE [ OF column_name [, ... ] ]
   DELETE
   TRUNCATE
```

第一个要注意的地方是触发器总是为一个表或者[①]一个视图被触发，并且调用一个函数。触发器有一个名称并且可以在事件之前或者之后发生。PostgreSQL 的妙处在于用户可以根据需要在单个表上有任意多个触发器。虽然这对专家级的 PostgreSQL 用户来说没什么可惊讶的，笔者还是要指出世界上仍在使用的很多昂贵的商业数据库引擎却做不到这一点。

如果在同一个表上有多于一个触发器，则会使用很多年前在 PostgreSQL 7.3 中引入的规则：触发器按照字母顺序被引发。首先，所有那些前触发器以字母顺序发生，然后 PostgreSQL 执行引发触发器的行操作并且继续按字母顺序执行后触发器。换句话说，触发器的执行顺序是绝对确定的，并且触发器的数量基本上是无限的。

触发器可以在实际修改发生之前或者之后修改数据。一般来说，这是一种验证数据并且在某些自定义限制被违背时报错的好办法。下面的例子展示了一个由 INSERT 子句引发的触发器，它会更改加入表的数据：

```
CREATE TABLE t_sensor (
        id              serial,
        ts              timestamp,
        temperature     numeric
);
```

我们的表只存储若干值，现在的目标是一插入行就调用一个函数：

```
CREATE OR REPLACE FUNCTION trig_func()
RETURNS trigger AS
$$
```

[①] 原文是“a table of a view”，应为笔误。

```
        BEGIN
                IF      NEW.temperature < -273
                THEN
                        NEW.temperature := 0;
                END IF;

                RETURN NEW;
        END;
$$ LANGUAGE 'plpgsql';
```

如前所述，触发器将总是调用一个函数，这允许用户精细地抽象代码。这里的重点是触发器函数必须返回 trigger。要访问将要插入的行，用户可以访问 NEW 变量。



INSERT 和 UPDATE 触发器总是提供一个 NEW 变量。UPDATE 和 DELETE 总是提供一个名为 OLD 的变量。那些变量包含将要修改的行。

在笔者的例子中，代码检查温度是否太低。如果温度太低，值就不行，它会被动态调整。为了确保能使用被修改的行，NEW 被返回。如果有第二个触发器在这一个之后被调用，下一个函数调用将看到被修改的行。

在下一步，创建这个触发器：

```
CREATE TRIGGER sensor_trig
    BEFORE INSERT ON t_sensor
    FOR EACH ROW
    EXECUTE PROCEDURE trig_func();
```

下面是该触发器的效果：

```
test=# INSERT INTO t_sensor (ts, temperature)
            VALUES ('2017-05-04 14:43', -300)
            RETURNING *;
 id |          ts          | temperature
----+---------------------+-------------
  1 | 2017-05-04 14:43:00 |           0
(1 row)

INSERT 0 1
```

如你所见，值已经被正确地调整，表中该温度显示为 0。

如果用户在使用触发器，用户应该意识到一点，触发器对其自身有很深的了解。它能访问若干变量，这些变量允许用户编写更精致的代码并且实现更好的抽象。

让我们先删除这个触发器：

```
test=# DROP TRIGGER sensor_trig ON t_sensor;
DROP TRIGGER
```

然后增加一个新函数：

```
CREATE OR REPLACE FUNCTION trig_demo()
RETURNS trigger AS
$$
        BEGIN
                RAISE NOTICE 'TG_NAME: %', TG_NAME;
                RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
                RAISE NOTICE 'TG_TABLE_SCHEMA: %',
                          TG_TABLE_SCHEMA;
                RAISE NOTICE 'TG_TABLE_NAME: %', TG_TABLE_NAME;
                RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
                RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
                RAISE NOTICE 'TG_OP: %', TG_OP;
                RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
                -- RAISE NOTICE 'TG_ARGV: %', TG_NAME;

                RETURN NEW;
   END;
$$ LANGUAGE 'plpgsql';
```

这里用到的所有变量都是预定义好的，并且默认就可用。我们的代码所做的只是显示它们以便能看到其内容：

```
CREATE TRIGGER demo_trigger
BEFORE INSERT ON t_sensor
FOR EACH ROW EXECUTE PROCEDURE trig_demo();

test=# INSERT INTO t_sensor (ts, temperature)
        VALUES ('2017-05-04 14:43', -300) RETURNING *;
NOTICE: TG_NAME: demo_trigger
NOTICE: TG_RELNAME: t_sensor
NOTICE: TG_TABLE_SCHEMA: public
NOTICE: TG_TABLE_NAME: t_sensor
NOTICE: TG_WHEN: BEFORE
NOTICE: TG_LEVEL: ROW
NOTICE: TG_OP: INSERT
NOTICE: TG_NARGS: 0
 id |          ts          | temperature
----+---------------------+-------------
```

```
 2 | 2017-05-04 14:43:00 |        -300
(1 row)

INSERT 0 1
```

这里读者可以看到，触发器知道它的名字，是为哪个表被触发等信息。如果用户想对多个表应用类似的动作，那些变量能帮助用户通过仅编写一个函数来避免重复代码，然后这个函数可以被用于所有感兴趣的表。

7.2.2　引入 PL/Perl

关于 PL/pgSQL 还有更多可说的。但是由于笔者只有有限的页数来涵盖这一主题，现在是时候进入下一种过程语言了。PL/Perl 已经被很多人接受作为处理字符串的理想语言。读者可能知道，Perl 以其字符串操纵能力而闻名，因此在这么多年后仍然相当流行。

要启用 PL/Perl，用户有两种选择：

```
[postgres@linuxpc ~]$ createlang plperl test
[postgres@linuxpc ~]$ createlang plperlu test
```

用户可以部署可信的和不可信的 Perl。如果两者都想要，用户必须两种语言都启用。

为了向读者展示 PL/Perl 如何工作，笔者已经实现了一个函数，它简单地解析一个 email 地址并且返回真或假。下面是该函数的定义：

```
test=# CREATE OR REPLACE FUNCTION verify email(text)
RETURNS boolean AS
$$
        if      ($_[0] =~ /^[a-z0-9.]+@[a-z0-9.-]+$/)
        {
                return true;
        }
        return false;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

一个文本参数被传递给该函数。在函数中，所有的输入参数可以通过使用$_访问。在这个例子中，先执行了正则表达式，然后函数返回。

可以像用任何其他语言编写的任何其他过程一样调用这个函数：

```
test=# SELECT verify_email('hs@cybertec.at');
 verify_email
--------------
```

```
 t
(1 row)

test=# SELECT verify_email('totally wrong');
 verify_email
--------------
 f
(1 row)
```

如果处于一个可信函数内部，记住不能载入包之类的东西。例如，如果想用 w 命令寻找单词，Perl 将在内部载入 utf8.pm，这当然是不被允许的。

1. 将 PL/Perl 用于数据类型抽象

正如本章中所述，PostgreSQL 中的函数相当通用并且能在很多不同的上下文中使用。如果用户想用函数来提升数据质量，可以使用 CREATE DOMAIN 子句：

```
test=# h CREATE DOMAIN
Command:     CREATE DOMAIN
Description: define a new domain
Syntax:
CREATE DOMAIN name [ AS ] data type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]

where constraint is:

[ CONSTRAINT constraint name ]
{ NOT NULL | NULL | CHECK (expression) }
```

在这个例子中，PL/Perl 函数将用来创建一个名为 email 的域，随后它可以被用作一种数据类型。

下面的列表展示了如何创建域：

```
test=# CREATE DOMAIN email AS text
            CHECK (verify_email(VALUE) = true);
CREATE DOMAIN
```

如前所述，域函数就像一种普通数据类型：

```
test=# CREATE TABLE t email (id serial, data email);
CREATE TABLE
```

如下例所示，这个 Perl 函数确保不会有违背检查的东西被成功地插入数据库中：

```
test=# INSERT INTO t_email (data)
            VALUES ('somewhere@example.com');
INSERT 0 1
test=# INSERT INTO t email (data)
            VALUES ('somewhere wrong example.com');
ERROR: value for domain email violates check
            constraint "email_check"
```

Perl 可能是一种做字符串处理的好选择，但是是否直接在数据库中用这种代码实现，终究还是由用户决定。

2. 在 PL/Perl 和 PL/PerlU 之间做决定

到目前为止，这段 Perl 代码还没有打开任何安全性相关的问题，因为笔者所用的都是正则表达式。现在的问题是，如果有人尝试在 Perl 函数内做些不好的事情会怎样？如前所述，PL/Perl 将简单地报错：

```
test=# CREATE OR REPLACE FUNCTION test_security()
RETURNS boolean AS
$$
        use strict;
        my $fp = open("/etc/password", "r");

        return false;
$$ LANGUAGE 'plperl';
ERROR: 'open' trapped by operation mask at line 3.
CONTEXT: compilation of PL/Perl function "test_security"
```

PL/Perl 将在尝试创建该函数时立即抱怨，一个错误将马上被显示出来。

如果用户真的想运行用 Perl 写的不可信代码，必须使用 PL/PerlU：

```
test=# CREATE OR REPLACE FUNCTION first_line()
RETURNS text AS
$$
        open(my $fh, '<:encoding(UTF-8)', "/etc/passwd")
            or elog(NOTICE, "Could not open file '$filename' $!");
        my $row = <$fh>;
        close($fh);

        return $row;
```

```
$$ LANGUAGE 'plperlu';
CREATE FUNCTION
```

基本上，这个过程和前一个是完全相同的，会返回一个字符串。但是，它被允许做所有的事情。二者唯一的区别是该函数被标记为 plperlu。

结果并不令人惊讶：

```
test=# SELECT first line();
           first line
---------------------------------
 root:x:0:0:root:/root:/bin/bash+

(1 row)
```

3. 使用 SPI 接口

有时，用户的 Perl 过程不得不做数据库工作。记住该函数是数据库连接的一部分。因此，再去创建一个数据库连接没有意义。为了与数据库对话，PostgreSQL 服务器基础设施提供了 SPI 接口，它是一个与数据库内部对话的 C 接口。所有帮助用户运行服务器端代码的过程语言都使用这个接口所披露的功能。PL/Perl 同样如此，在本节中，读者将学到如何使用 Perl 包装这种 SPI 接口。

用户可能想做的最重要的事情是运行 SQL 并且检索取得的行数。spi_exec_query 函数就负责这种工作。第二个参数是实际想要检索的行数。为了保持简单，笔者决定检索所有的行：

```
test=# CREATE OR REPLACE FUNCTION spi sample(int)
RETURNS void AS
$$
        my $rv = spi_exec_query("
                            SELECT *
                            FROM generate_series(1, $_[0])", $_[0]);
        elog(NOTICE, "rows fetched: " . $rv->{processed});
        elog(NOTICE, "status: " . $rv->{status});

        return;
$$ LANGUAGE 'plperl';
```

SPI 会漂亮地执行这个查询并且显示行数。这里的重点是所有的存储过程语言都提供一种方法来发送日志消息。在 PL/Perl 中，这个函数叫作 elog，并且接受两个参数。第一个参数定义消息的重要度（INFO、NOTICE、WARNING、ERROR 等），而第二个参数

包含实际的消息。

下面的消息显示查询返回的内容：

```
test=# SELECT spi_sample(9);
NOTICE:  rows fetched: 9
NOTICE:  status: SPI_OK_SELECT
 spi_sample
------------

(1 row)
```

4. 将 SPI 用于集合返回函数

在很多情况下，用户不只想执行某个 SQL 就完了。在大部分情况下，一个过程将在结果上循环并且用结果做一些事情。下面的例子将展示用户如何在查询的输出上循环。此外，笔者决定对这个例子做一点补充并且让该函数返回一种组合类型。在 Perl 中使用组合类型很容易，因为用户可以简单地把数据填入到一个哈希表并且返回。return_next 函数将会逐步构建起结果集，直到函数被一个返回语句终止。

这个列表中的例子产生一个包含随机值的表：

```
CREATE TYPE random_type AS (a float8, b float8);

CREATE OR REPLACE FUNCTION spi_srf_perl(int)
RETURNS setof random_type AS
$$
        my $rv = spi_query("SELECT random() AS a, random() AS b
                              FROM generate_series(1, $_[0])");
        while (defined (my $row = spi_fetchrow($rv)))
        {
                elog(NOTICE, "data: " . $row->{a}
                          . " / " . $row->{b});
                return_next({
                        a_col => $row->{a},
                        b_col => $row->{b}
                });
        }
        return;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

首先，spi_query 函数被执行并且使用 spi_fetchrow 函数开始一个循环。在循环内，组合类型将被组装并且被填入结果集中。

不出所料，该函数将返回一组随机值：

```
test=# SELECT * FROM spi_srf_perl(3);
NOTICE: data: 0.154673356097192 / 0.278830723837018
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.615888888947666 / 0.632620786316693
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.910436692181975 / 0.753427186980844
CONTEXT: PL/Perl function "spi_srf_perl"
      a_col       |       b_col
------------------+-------------------
 0.154673356097192 | 0.278830723837018
 0.615888888947666 | 0.632620786316693
 0.910436692181975 | 0.753427186980844
(3 rows)
```

记住，集合返回函数必须被实现，这样整个结果集将被存储在内存中。

5. 在 PL/Perl 中的转义以及支持函数

到目前为止，我们只使用了整数，因此 SQL 注入或特殊表名不会是问题。如果要面对这类问题，有下列函数可用。

- quote_literal：返回一个按字符串字面值引用的字符串。
- quote_nullable：对一个字符串加引用。
- quote_ident：对 SQL 标识符（对象名等）加引用。
- decode_bytea：解码一个 PostgreSQL 字节数组域。
- encode_bytea：编码数据并且将其转变成一个字节数组。
- encode_literal_array：编码一个字面值数组。
- encode_typed_literal：把一个 Perl 变量转换成作为第二个参数传入的数据类型的值，并且返回该值的字符串表示。
- encode_array_constructor：把被引用数组的内容转变成数组构造器格式中的字符串。
- looks_like_number：如果一个字符串看起来像一个数字，则返回真。
- is_array_ref：如果某个东西是一个数组引用，则返回真。

这些函数总是可用，并且可以被直接调用而不必包括任何库。

6. 在函数调用之间共享数据

有时有必要在多次调用之间共享数据，这种基础设施也有方法来实际做到这一点。在 Perl 中，一个哈希表被用来存储任何需要的数据：

```
CREATE FUNCTION perl_shared(text)
RETURNS int AS
$$
    if      (!defined $_SHARED{$_[0]})
    {
           $_SHARED{$_[0]} = 0;
    }
    else
    {
           $_SHARED{$_[0]}++;
    }

    return $_SHARED{$_[0]};
$$ LANGUAGE 'plperl';
```

一旦我们发现传递给函数的键不存在，$_SHARED 变量就将被初始化为 0。对于每一次其他调用，计数器会被加 1，最后给我们下面的输出：

```
test=# SELECT perl_shared('some_key') FROM generate_series(1, 3);
 perl_shared
-------------
           0
           1
           2
(3 rows)
```

对更复杂语句的情况，开发者通常不知道函数将被以何种顺序调用，因此要牢记在大部分情况下不能依赖于执行顺序。

7. 用 Perl 编写触发器

每一种随 PostgreSQL 核心发行的存储过程语言都允许用户用其编写存储过程，这同样也适用于 Perl。由于篇幅限制，笔者决定不包括用 Perl 编写的触发器例子，而是将读者指引到官方的 PostgreSQL 文档：https://www.postgresql.org/docs/9.6/static/plperl-triggers.html。

基本上，用 Perl 编写触发器和用 PL/pgSQL 没什么区别。所有预定义的变量也同样可用，至于返回值，规则适用于每一种存储过程语言。

7.2.3 引入 PL/Python

如果用户恰好不是一个 Perl 专家，PL/Python 可能会更合适。很久以前，Python 就已经成为 PostgreSQL 基础设施的一部分，因此它是一种可靠的、久经考验的实现。

在谈到 PL/Python 时，有一件事必须记住，PL/Python 只能用作一种不可信语言。从安全性的角度而言，有必要始终牢记这一点。

要启用 PL/Python，用户可以从其命令行运行下面的命令。test 是想要与 PL/Python 一起使用的数据库名：

```
createlang plpythonu test
```

一旦该语言被启用，就已经可以编写代码了。

当然，用户可以选择使用 CREATE LANGUAGE 子句。还要记住，为了使用服务器端语言，需要包含那些语言的 PostgreSQL 包（postgresql-plpython-9.6 等）。

1. 编写简单的 PL/Python 代码

在本节中，读者将学到编写简单的 Python 过程。这里讨论的例子非常简单：如果用户正在奥地利通过汽车拜访客户，用户可以每公里扣除 42 欧分作为开支来降低用户的所得税。所以该函数要做的事情是得到公里数并且返回可以从税金账单中扣除的钱数。代码如下：

```
CREATE OR REPLACE FUNCTION calculate deduction(km float)
RETURNS numeric AS
$$
        if      km <= 0:
                elog(ERROR, 'invalid number of kilometers')
        else:
                return km * 0.42
$$ LANGUAGE 'plpythonu';
```

该函数确保只接受正值。最后，结果被计算出来并且返回。如你所见，Python 函数被传递给 PostgreSQL 的方式实际上与 Perl 或 PL/pgSQL 没什么不同。

2. 使用 SPI 接口

和所有过程语言一样，PL/Python 也可以访问 SPI 接口。下面的例子展示了如何将数字加起来：

```
CREATE FUNCTION add_numbers(rows_desired integer)
        RETURNS integer AS
$$
        mysum = 0

        cursor = plpy.cursor("SELECT * FROM
                   generate series(1, %d) AS id" % (rows desired))

        while True:
                rows = cursor.fetch(rows_desired)
                if not rows:
                        break
                for row in rows:
                        mysum += row['id']
        return mysum
$$ LANGUAGE 'plpythonu';
```

在试验这个例子时，要确保对游标的调用确实在单行中。Python 依赖于缩进，因此代码由一行还是两行组成会完全不同。

一旦游标被创建，我们可以在其上进行循环并且把那些数字加起来。那些行中的列可以通过使用列名很容易地引用。

调用该函数将返回想要的结果：

```
test=# SELECT add numbers(10);
 add numbers
-------------
          55
(1 row)
```

如果用户想要检查一个 SQL 语句的结果集，PL/Python 提供了多种函数从结果中检索更多信息。再一次，这些函数都是对 SPI 提供的 C 层函数的包装。

下面的函数会更近距离地检查结果：

```
CREATE OR REPLACE FUNCTION result_diag(rows_desired integer)
        RETURNS integer AS
$$
        rv = plpy.execute("SELECT *
               FROM generate series(1, %d) AS id" % (rows desired))
        plpy.notice(rv.nrows())
        plpy.notice(rv.status())
        plpy.notice(rv.colnames())
        plpy.notice(rv.coltypes())
```

```
        plpy.notice(rv.coltypmods())
        plpy.notice(rv.__str__())

        return 0
$$ LANGUAGE 'plpythonu';
```

nrows 函数将显示行数。status 函数告诉我们是否一切正常。colnames 函数返回列的一个列表。coltypes 函数返回结果集中数据类型的对象 ID，23 是整数的内部编号：

```
test=# SELECT typname FROM pg type WHERE oid = 23;
 typname
---------
 int4
(1 row)
```

然后是 typmod。考虑 varchar(20)这样的类型，其中的配置部分说明了类型有什么样的 typmod。

最后有一个函数把全部信息作为一个字符串返回用于调试目的，调用这个函数将返回下列结果：

```
test=# SELECT result_diag(3);
NOTICE:  3
NOTICE:  5
NOTICE:  ['id']
NOTICE:  [23]
NOTICE:  [-1]
NOTICE:  <PLyResult status=5 nrows=3 rows=[{'id': 1},
                  {'id': 2}, {'id': 3}]>
 result_diag
-------------
           0
(1 row)
```

在 SPI 接口中有更多的函数来帮助执行 SQL。

3. 处理错误

有时，用户可能必须捕捉错误。当然，这在 Python 中也是可以做到的。下面的例子展示了如何捕获错误：

```
CREATE OR REPLACE FUNCTION trial error()
        RETURNS text AS
```

```
$$
    try:
        rv = plpy.execute("SELECT surely a syntax error")
    except plpy.SPIError:
        return "we caught the error"
    else:
        return "all fine"
$$ LANGUAGE 'plpythonu';
```

用户可以使用正常的 try/except 块并且访问 plpy 来处理想要捕获的错误，然后函数可以正常返回而不毁掉事务：

```
test=# SELECT trial_error();
     trial_error
---------------------
 we caught the error
(1 row)
```

记住，PL/Python 能完全访问 PostgreSQL 的内部。因此，它还可能暴露所有类型的错误给用户的过程。这里有一个例子：

```
except spiexceptions.DivisionByZero:
    return "found a division by zero"
except spiexceptions.UniqueViolation:
    return "found a unique violation"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
```

在 Python 中捕捉错误确实很容易并且能帮助防止函数失败。

7.3　改进存储过程的性能

到目前为止，读者已经见过了如何用多种语言编写基本的存储过程以及触发器。当然，有更多的语言被支持。一些最突出的包括 PL/R（R 是一种强大的统计包）以及 PL/v8（基于 Google JavaScript 引擎）。不过，那些语言超过了本章的范围（先不管它们是否有用）。

本节将聚焦于提高存储过程的性能。可以在若干领域加速存储过程的处理：

- 减少调用次数。
- 使用缓存计划。
- 给优化器提示。

在本章中将讨论这 3 个主要领域。

● 减少函数调用次数

在很多情况下，性能不好是因为函数被过于频繁地调用。有一点再怎么强调也不为过，过于频繁地调用是性能不好的主因。在创建一个函数时，用户可以选择 3 种类型的函数：volatile、stable 以及 immutable。这里有一个例子：

```
test=# SELECT random(), random();
      random       |      random
-------------------+-------------------
 0.276252629235387 | 0.710661871358752
(1 row)
test=# SELECT now(), now();
              now              |              now
-------------------------------+-------------------------------
 2016-12-16 12:57:17.135751+01 | 2016-12-16 12:57:17.135751+01
(1 row)

test=# SELECT pi();
        pi
------------------
 3.14159265358979
(1 row)
```

volatile 函数意味着该函数不能被优化掉，它不得不被一次又一次地执行。volatile 函数还可能是为什么没有使用一个特定索引的原因。默认情况下，每一个函数都被认为是 volatile。stable 函数在同一个事务中将总是返回相同的数据。它可以被优化并且调用可以被移除。now 函数是 stable 函数的一个绝佳例子，在同一个事务中它总是返回相同的数据。

immutable 函数是黄金标准，因为它们允许进行大部分优化，那是因为它们总是对给定的相同输入返回相同的结果。作为优化函数的第一步，要总是确保在定义它们时用 volatile、stable 或者 immutable 正确地标记它们。

1. 使用缓存计划

在 PostgreSQL 中，查询被分成 4 个阶段执行。

● **解析器：**检查语法。

● **重写系统：**处理规则等。

● **优化器/规划器：**优化查询。

● **执行器：**执行规划器提供的计划。

如果查询很短，前 3 步相对于实际执行时间来说更耗时。因此，有必要缓存执行计划。PL/pgSQL 基本上会在幕后自动缓存所有的计划，用户无须操心。PL/Perl 和 PL/Python 则会让用户选择。SPI 接口提供了处理和运行预备查询的函数，因此程序员可以选择查询是否应该被预备。在查询很长的情况下，可能确实有必要使用非预备查询——但短查询通常应该总是被预备以减小内部开销。

2. 为函数指派代价

从优化器的角度来看，一个函数基本上就像是一个操作符。PostgreSQL 也将以同样的方式处理代价，就好像它是一个标准的操作符。只是问题在于：加两个数通常比使用某个 PostGIS 提供的函数做海岸线相交代价更低。原因就在于优化器不知道一个函数是便宜还是昂贵。幸运的是，我们可以告诉优化器让函数变得更便宜或者更昂贵：

```
test=# h CREATE FUNCTION
Command:     CREATE FUNCTION
Description: define a new function
Syntax:
CREATE [ OR REPLACE ] FUNCTION
...
    | COST execution_cost
    | ROWS result_rows
   ...
```

COST 参数表示用户的操作符实际比标准操作符贵多少。它是一个用于 cpu_operator_cost 的乘数而不是一个静态值。通常，除非函数由 C 写成，默认值是 100。

第二个参数是 ROWS 函数。PostgreSQL 默认假设集合返回函数将返回 1000 行，因为系统无法精确地确定到底有多少行。ROWS 参数允许开发者告诉 PostgreSQL 预期的行数。

7.4　使用存储过程

在 PostgreSQL 中，存储过程可以被用于很多事情。在本章中，读者已经学到了 CREATE DOMAIN 子句等，但还可以创建用户自己的操作符、类型转换甚至排序规则。

在本节中，读者将看到如何创建一个简单的类型转换以及如何用它来获益。为了定义类型转换，考虑一下 CREATE CAST 子句：

```
test=# h CREATE CAST
Command:     CREATE CAST
Description: define a new cast
```

```
Syntax:
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function name (argument type [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

使用它非常简单。用户只要告诉 PostgreSQL 它会调用哪个过程把何种类型转换成用户想要的数据类型即可。

在标准的 PostgreSQL 中，用户无法把一个 IP 地址转换成布尔值。因此，这就是一个很好的例子。首先必须定义存储过程：

```
CREATE FUNCTION inet_to_boolean(inet)
RETURNS boolean AS
$$
        BEGIN
                RETURN true;
        END;
$$ LANGUAGE 'plpgsql';
```

为了简单，它返回真。不过，用户可以使用任何语言的任何代码（当然）来做真正的转换。

在下一步，已经可以定义类型转换：

```
CREATE CAST (inet AS boolean)
    WITH FUNCTION inet to boolean(inet)
    AS IMPLICIT;
```

第一件事是告诉 PostgreSQL，我们想要把 inet 转换成 boolean。然后该函数被列出，并且告诉 PostgreSQL 要隐式转换。

这种转换是一种直接的处理，我们可以测试这个转换：

```
test=# SELECT '192.168.0.34'::inet::boolean;
 bool
------
```

```
 t
(1 row)
```

基本上，同样的逻辑也可以被应用于定义排序规则。同样，一个存储过程还可以被用来执行任何需要完成的工作：

```
test=# h CREATE COLLATION
Command:     CREATE COLLATION
Description: define a new collation
Syntax:
CREATE COLLATION name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype ]
)
CREATE COLLATION name FROM existing_collation
```

7.5　总　　结

在本章中，读者学到了如何编写存储过程。在一番理论性的介绍后，我们的注意力集中在 PL/pgSQL 的某些特性上。此外，读者还学到了如何使用 PL/Perl 以及 PL/Python，它们是 PostgreSQL 提供的两种重要语言。当然，还有更多的语言可用。但是，由于本书的范围（和篇幅）限制，无法详细介绍它们。如果读者想要了解更多，可以看看下面的网站：https://wiki.postgresql.org/wiki/PL_Matrix。

在第 8 章中，读者将学习有关 PostgreSQL 安全性的内容，以及从总体上学习如何管理用户和权限。此外，读者还将学习有关网络安全的内容。

第 8 章　管理 PostgreSQL 安全性

第 7 章的内容全都与存储过程和编写服务器端代码有关。在向读者介绍了很多重要主题之后，现在是时候将我们的焦点转向 PostgreSQL 的安全性了。读者将学到如何保护服务器以及配置权限。

本章将涵盖下列主题：

- 配置网络访问。
- 管理认证。
- 处理用户和角色。
- 配置数据库安全性。
- 管理方案、表和列。
- 行级安全性。

在本章结束时，读者将能够配置和管理 PostgreSQL 的安全性[①]。

8.1　管理网络安全性

在进入真实世界的实际例子之前，笔者想把读者的注意力稍稍转向我们将要处理的多个安全性层次。在处理安全性时，有必要把这些层次记在心里以便有条理地解决与安全性相关的问题。

下面是笔者心目中的安全性层次模型。

- 绑定地址：postgresql.conf 文件中的 listen_addresses。
- 基于主机的访问控制：pg_hba.conf 文件。
- 实例级权限：用户、角色、数据库创建、登录以及复制。
- 数据库级权限：连接、创建方案等。
- 方案级权限：使用方案以及在方案中创建对象。
- 表级权限：选择、插入、更新等。
- 列级权限：允许或者限制对列的访问。

[①] 原文此处是“At the end of the chapter, you will be able to write good and efficient procedures.”，这和第 7 章对应位置的句子完全相同，显然这一章的内容是关于安全性而不是存储过程，因此原文应为笔误。这里译者按照原作者的风格加上了一个类似的句子。

- 行级安全性：限制对行的访问。

为了读取一个值，PostgreSQL 必须确定用户在每一个级别上具有足够的权限，整个权限链必须正确。

8.1.1　理解绑定地址和连接

在配置 PostgreSQL 服务器时，首先要做的事情之一是定义远程访问，默认 PostgreSQL 不接受远程连接。这里的重点是 PostgreSQL 甚至都没有机会拒绝连接，因为它根本没有在端口上监听。如果用户尝试连接，错误消息实际上将由操作系统返回，因为 PostgreSQL 根本就没有关心远程连接。

假定有一台数据库服务器在 192.168.0.123 上使用默认配置，将发生下面的效果：

```
iMac:~ hs$ telnet 192.168.0.123 5432
Trying 192.168.0.123...
telnet: connect to address 192.168.0.123: Connection refused
telnet: Unable to connect to remote host
```

telnet 尝试在端口 5432 上创建一个连接，并且将立即被远端的机器拒绝。从外部来看，就好像 PostgreSQL 根本没有运行。

成功的关键可以在 postgresql.conf 文件中找到：

```
# - Connection Settings -

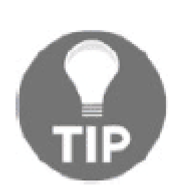
#listen addresses = 'localhost'
            # what IP address(es) to listen on;
           # comma-separated list of addresses;
           # defaults to 'localhost'; use '*' for all
           # (change requires restart)
```

listen_addresses 设置将告诉 PostgreSQL 要在哪些地址上监听。从技术上来讲，那些地址就是绑定地址。那实际意味着什么？假定用户的机器上有 4 块网卡，用户可以在那些 IP 地址中的 3 个上监听。PostgreSQL 会考虑对那 3 块网卡的请求并且不在第 4 块上监听，在第 4 块网卡上端口根本就是关闭的。



用户必须将其服务器的 IP 地址而不是客户端的 IP 放入 listen_addresses。

如果在其中放入一个'*'，PostgreSQL 将在机器的每一个 IP 上监听。



记住更改 listen_addresses 要求一次 PostgreSQL 服务重启。如果没有重启，它不能被即时改变。

不过，还有更多与连接管理相关的设置非常有必要去理解：

```
#port = 5432
                # (change requires restart)
max_connections = 100
                # (change requires restart)
# Note: Increasing max connections costs ~400 bytes of
#       shared memory per
# connection slot, plus lock space
#           (see max_locks_per_transaction).
#superuser_reserved_connections = 3
                # (change requires restart)
#unix_socket_directories = '/tmp'
                # comma-separated list of directories
                # (change requires restart)
#unix socket group = ''
                # (change requires restart)
#unix_socket_permissions = 0777
                # begin with 0 to use octal notation
                # (change requires restart)
```

首先，PostgreSQL 对一个单一 TCP 端口（默认值是 5432）监听。记住 PostgreSQL 将只在单个端口上监听。只要一个请求进来，postmaster 将会派生并且创建一个新进程来处理该连接。默认情况下允许最多 100 个普通连接。此外，为超级用户保留了 3 个额外的连接。这就意味着可以有 100 个连接外加 3 个超级用户或者 103 个超级用户连接。注意那些连接相关的设置也将需要一次重启，其原因是在共享内存中为连接分配的是静态数量的内存，这无法被即时更改。

1. 检查连接和性能

在笔者做咨询时，很多人问到提高连接限制是否将会对总体性能造成影响。答案是：影响不会太大（总是会有一些上下文切换等造成的开销）。有多少连接基本上不会造成什么区别。不过，真正造成不同的是打开快照的数量。打开快照（不是连接）的数量越多，服务器端的开销就越大。换句话说，用户增加 max_connections 的代价很低。

如果读者对一些真实世界的数据感兴趣，考虑看看笔者的一篇老博文：http://www.cybertec.at/max_connections-performance-impacts/。

2. 不使用 TCP 的世界

在某些情况下，用户可能不想使用网络，这常常发生在数据库与本地应用交互的场

景中。也许，用户的 PostgreSQL 数据库是与应用一起发售的，或者也许用户不想冒使用网络的风险：在这种情况下，就需要 Unix 套接字。Unix 套接字是一种无须网络的通信方法。用户的应用可以通过一个 Unix 套接字进行本地的连接而不对外暴露任何东西。

不过，用户所需要的只是一个目录。PostgreSQL 默认将使用/tmp 目录[①]。但是，如果每台机器上运行不止一个数据库服务器，每一个数据库服务器都将需要一个单独的数据目录。

除了安全性，还有很多不使用网络的理由。其中之一是性能。使用 Unix 套接字比通过环回设备（127.0.0.1）快很多。如果读者对此感到惊讶，别担心——很多人都和你一样。不管怎样，如果只运行非常小的查询，真实网络连接的开销不应该被低估。

为了向读者展示这到底意味着什么，笔者在这里包括了一个简单的基准。

笔者已经创建了一个 script.sql 文件。这是一个简单的脚本，它只是创建一个随机数并且选择这个数。因此它可能是最简单的语句了，没有什么比只取一个数更简单。

如此，让我们在一台普通的笔记本上运行这个简单的基准。为了运行基准，笔者已经写好了一个小东西 script.sql，它将会被该基准用到：

```
[hs@linuxpc ~]$ cat /tmp/script.sql
SELECT 1
```

然后就可以简单地运行 pgbench 来一遍又一遍地执行这个 SQL。-f 选项允许把脚本的名字传给 pgbench。-c 10 表示我们想要 10 个并发连接并且连接会活跃 5 秒（-T 5）。这个基准被作为 postgres 用户运行并且被假定使用 postgres 数据库，该数据库应该默认存在。注意下面的例子将能在 RHEL 衍生系统上运行，基于 Debian 的系统将使用不同的路径：

```
[hs@linuxpc ~]$ /usr/pgsql-9.6/bin/pgbench -f /tmp/script.sql
              -c 10 -T 5
              -U postgres postgres 2> /dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 871407
latency average = 0.057 ms
tps = 174278.158426 (including connections establishing)
tps = 174377.935625 (excluding connections establishing)
```

[①] 原文是"command"，应为笔误。

如你所见，没有向 pgbench 传递主机名，因此这个工具本地连接到 Unix 套接字并且尽可能快地运行该脚本。在这个四核的 Intel 机器上，系统能够达到每秒 174000 个事务。

如果加上-h localhost 会发生什么？

```
[hs@linuxpc ~]$ /usr/pgsql-9.6/bin/pgbench -f /tmp/script.sql
              -h localhost -c 10 -T 5
              -U postgres postgres 2> /dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 535251
latency average = 0.093 ms
tps = 107000.872598 (including connections establishing)
tps = 107046.943632 (excluding connections establishing)
```

吞吐量将跌落到每秒 107000 个事务。区别很明显与网络开销有关。



通过使用-j 选项（分派给 pgbench 的线程数），用户可以把更多的事务挤出系统外。但是，在笔者的情况中这并不会改变该基准的总体结果。在其他的测试中，它确实会造成改变，因为如果没有提供足够的 CPU 能力，pgbench 可能会是一个真正的瓶颈。

如你所见，网络可能不只是一种安全性问题，还可能是一种性能问题。

8.1.2　管理 pg_hba.conf

在配置好绑定地址之后，就可以转到下一个级别。pg_hba.conf 文件将告诉 PostgreSQL 如何认证来自网络的用户。通常，pg_hba.conf 文件项具有下面的格式：

```
# local      DATABASE USER METHOD [OPTIONS]
# host       DATABASE USER ADDRESS METHOD [OPTIONS]
# hostssl    DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnossl DATABASE USER ADDRESS METHOD [OPTIONS]
```

可以在 pg_hba.conf 文件中放入以下 4 类规则。

- local：可以被用来配置本地 Unix 套接字连接。
- host：可以被用于 SSL 和非 SSL 连接。
- hostssl：只可用于 SSL 连接。为了使用该选项，服务器中必须编译有 SSL，

PostgreSQL 的预包装版本就编译有 SSL。此外，服务器启动时必须在 postgresql.conf 文件中已经设置有 ssl = on。

- hostnossl：用于非 SSL 连接。

一个规则列表可以放在 pg_hba.conf 文件中。这里是一个例子：

```
# TYPE   DATABASE     USER       ADDRESS            METHOD
 local    all         all                            trust
 host     all         all        127.0.0.1/32        trust
 host     all         all        ::1/128             trust
```

可以看到 3 条简单的规则。local 记录说明要访问所有数据库的来自本地 Unix 套接字的所有用户都被信任。trust 方法意味着不会有口令被发送到服务器并且人们可以直接登录。其他两个规则说明将同样的方法应用在来自本地主机 127.0.0.1 和::1/128（这是一个 IPv6 连接）的连接上。

由于不用口令连接肯定不是远程访问最好的选择，PostgreSQL 提供了多种认证方法以便更灵活地配置 pg_hba.conf 文件。下面是可能的认证方法列表。

- trust：允许不提供密码的认证。要求用户必须在 PostgreSQL 端可用。
- reject：连接将被拒绝。
- md5 和 password：连接可以使用口令创建。md5 意味着口令在线缆上发送时被加密。在 password 的情况下，凭证被以明文发送，在现代系统中不应该这样做。
- GSS 和 SSPI：这种方法使用 GSSAPI 或 SSPI 认证。这种方法只对 TCP/IP 连接可用。其想法是允许单点登录。
- ident：这种方法通过联系客户端上的 Ident 服务器获得客户端的操作系统用户名，并且检查它是否匹配被请求的数据库用户名。
- peer：假定用于作为 abc 登录 Unix。如果启用 peer，用户就只能作为 abc 登录 PostgreSQL。如果用户尝试更改用户名，它将被拒绝。这里好的一点是 abc 将不需要口令进行认证。其思想是只有数据库管理员能登录 Unix 系统上的数据库，而其他只有口令或者同一机器上 Unix 账户的人则不能登录。这只对本地连接有用。
- pam：它使用可插拔认证模块（PAM）。如果想要使用一种不是 PostgreSQL 自带的认证方式，这就特别重要。要使用 PAM，在 Linux 系统上创建文件 /etc/pam.d/postgresql，并且把计划要使用的 PAM 模块放在这个配置文件中。通过使用 PAM，甚至可以使用不常见的组件进行认证。不过，它也能被用来连接到活动目录等服务。
- ldap：这种配置允许用户使用轻量级目录访问协议（LDAP）进行认证。注意 PostgreSQL 将只请求 LDAP 认证，如果一个用户只存在于 LDAP 上但不存在于

PostgreSQL 这边，用户就不能登录。还要注意 PostgreSQL 必须知道 LDAP 服务器在哪里。所有这些信息都必须被存放在 pg_hba.conf 文件中，具体可见官方文档：https://www.postgresql.org/docs/9.6/static/auth-methods.html#AUTH-LDAP。

- radius：远程认证拨入用户服务（RADIUS）是一种实现单点登录的方法。再一次，参数使用配置选项传入。
- cert：这种认证方法使用 SSL 客户端证书来执行认证，因此只有使用 SSL 才能使用它。这里的优势是不必要发送口令。证书的 CN 属性将被与请求的数据库用户名比较，如果它们匹配，登录将被允许。可以用一个映射来允许用户映射。

可以简单地一个接一个地列出规则。重点是规则的顺序确实会产生区别，如下例所示：

```
host    all         all         192.168.1.0/24         md5
host    all         all         192.168.1.54/32        reject
```

当 PostgreSQL 读取 pg_hba.conf 文件时，它将使用匹配上的第一条规则。因此，如果我们的请求来自于 192.168.1.54，第一条规则将总是在我们走到第二条规则之前就匹配上。这意味着如果口令和用户正确，192.168.1.54 将能登录。因此，第二条规则是没有意义的。

如果想要排除这个 IP，就应该交换这两条规则。

- 处理 SSL

PostgreSQL 允许用户加密服务器和客户端之间的传输。加密非常有益，特别是在进行长距离通信时。SSL 提供了一种简单且安全的方式来确保无人能监听通信。在本节中，读者将学到如何设置 SSL。

第一件要做的事情是在服务器启动时把 postgresql.conf 文件中的 ssl 参数设置为 on。下一步，可以把 SSL 证书放到$PGDATA 目录中。如果不想证书放在某个其他目录中，请更改下列参数：

```
#ssl_cert_file = 'server.crt'    # (change requires restart)
#ssl_key_file = 'server.key'     # (change requires restart)
#ssl_ca_file = ''                # (change requires restart)
#ssl_crl_file = ''               # (change requires restart)
```

如果想要使用自签名证书，执行下列步骤：

```
openssl req -new -text -out server.req
```

回答 OpenSSL 提出的问题。确保输入本地主机名作为 common name。可以将口令留空。这个调用将生成一个被口令保护的密钥，它将不会接受短于 4 个字符的口令。要移

除口令（如果要自动启动服务器就必须这样做），运行命令：

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

输入旧的口令解锁现有的密钥。现在执行下面的命令将该证书转变成一个自签名证书并且把密钥和证书复制到服务器查找它们的地方：

```
openssl req -x509 -in server.req -text
     -key server.key -out server.crt
```

最后，确保这些文件有正确的权限集：

```
chmod og-rwx server.key
```

一旦正确的规则被放入 pg_hba.conf 文件，就可以使用 SSL 来连接服务器。要验证的确在使用 SSL，考虑检查 pg_stat_ssl 函数。它将显示每一个连接以及它们是否使用 SSL，并且它将提供有关加密的重要信息：

```
test=# d pg_stat_ssl
   View "pg_catalog.pg_stat_ssl"
   Column     |   Type   | Modifiers
-------------+---------+-----------
 pid          | integer |
 ssl          | boolean |
 version      | text    |
 cipher       | text    |
 bits         | integer |
 compression  | boolean |
 clientdn     | text    |
```

如果一个进程的 ssl 域包含真，那么 PostgreSQL 就已经开始使用 SSL：

```
postgres=# select * from pg_stat_ssl;
-[ RECORD 1 ]---------------------------
pid          | 20075
ssl          | t
version      | TLSv1.2
cipher       | ECDHE-RSA-AES256-GCM-SHA384
bits         | 256
compression  | f
clientdn     |
```

8.1.3　处理实例级安全性

到目前为止，我们已经配置了绑定地址并且我们已经告诉 PostgreSQL 对哪些 IP 范围使用哪种认证方法。至今，配置都只与网络相关。

下一步，可以把注意力转到实例级的权限。最重要的一点是 PostgreSQL 中的用户都存在于实例级。如果创建一个用户，它不只是在一个数据库中可见——它可以被所有的数据库看见。一个用户可能只有访问单一数据库的权限，但本质上用户是被创建在实例级别。

对于新接触 PostgreSQL 的人，还有一件事情应该记住——用户和角色是同一种东西。CREATE ROLE 和 CREATE USER 子句有不同的默认值（正确地说，唯一的区别是角色默认没有 LOGIN 属性），但总而言之，用户和角色是相同的。因此，CREATE ROLE 和 CREATE USER 子句支持非常相似的语法：

```
test=# h CREATE USER
Command:     CREATE USER
Description: define a new database role
Syntax:
CREATE USER name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role name [, ...]
    | SYSID uid
```

让我们逐个讨论这些语法元素。第一点是用户可以是一个超级用户或者一个普通用

户。如果某个人被标记为超级用户，就没有普通用户必须面对的任何限制。超级用户可以根据其意愿删除对象（数据库等）。

下一个重点是需要有实例级权限来创建新数据库。注意当某个人创建一个数据库时，这个用户将自动成为该数据库的拥有者。规则是：创建者总是自动地成为对象的拥有者（除非像 CREATE DATABASE 子句中特别指定其他人作为拥有者）。好处是对象拥有者还可以再次删除对象。



CREATEROLE/NOCREATEROLE 子句定义某个人是否被允许创建新用户/角色。

再下一个重点是 INHERIT/NOINHERIT 子句。如果 INHERIT 子句被设置（默认值），用户可以从某个其他用户继承权限。使用继承的权限允许把角色用作一种抽象权限的方法。例如，可以创建一个会计员的角色并且让很多其他角色从会计员继承。其想法在于，即使有很多人都在做会计工作，也只需要告诉 PostgreSQL 一次会计员能做什么。

LOGIN/NOLOGIN 子句定义一个角色是否被允许登录实例。注意，LOGIN 子句并不足以实际地连接到一个数据库。要做到这一点，还需要更多权限。目前，试图让它进入实例中，LOGIN 子句基本上是实例中所有数据库的大门。让我们回到我们的例子：会计员可能被标记为 NOLOGIN，因为我们想让人们用他们的真实名称登录。所有的会计（即 joe 和 jane）可以被标记为 LOGIN，但能从会计员角色继承所有权限。

如果用户计划运行 PostgreSQL 的流复制，用户可以作为超级用户做所有的事务日志流式传送。但是，从安全性的角度来看，不推荐这样做。为了确保不需要使用超级用户流式传送 xlog，PostgreSQL 允许把复制权限给予普通用户，之后它就能被用来做流式传送。

正如稍后在本章中将要看到的那样，PostgreSQL 提供一种称为行级安全性的特性。其思想是可以从一个用户的视野中排除行。如果用户被明确设定为可以绕过 RLS，将这个值设置为 BYPASSRLS。默认值是 NOBYPASSRLS。

有时有必要限制一个用户允许的连接数。CONNECTION LIMIT 允许做这样的限制。注意总体上连接数绝不会超过 postgresql.conf 文件中的设置（max_connections）。但是可以把特定用户限制到较低的值。

默认情况下，PostgreSQL 将在系统表中加密存储口令，这是一种好的默认行为。不过，假定用户正在做一次培训课程。10 个学生参加并且每个人都被连接到系统中。我们可以 100%肯定这些人中的一个将偶尔忘记他/她的口令。由于用户的设置中安全性并非关键，用户可能决定以明文存储口令，这样可以很容易地查找口令并且把它告诉学生。如果正在测试软件，这种特性也派得上用场。

我们常常会知道某人将很快离开我们的机构。VALID UNTIL 子句允许我们在一个特

定用户的账户过期后自动锁住他/她。

IN ROLE 列出一个或者更多现有的角色，新角色将被立即加入它们作为新成员，这有助于避免额外的手工步骤。IN ROLE 的一种替代方法是 IN GROUP。

ROLE 子句将定义被自动作为新角色成员的角色。ADMIN 子句和 ROLE 子句相同，但是增加了 WITH ADMIN OPTION。

最后，可以使用 SYSID 子句为这个用户指定一个特定的 ID（类似于某些 Unix 管理员对操作系统级别用户名所作的事情）。

- 创建和修改用户

在这些理论介绍之后，让我们实际创建用户并且看看在实际的例子中如何使用它们：

```
test=# CREATE ROLE bookkeeper NOLOGIN;
CREATE ROLE
test=# CREATE ROLE joe LOGIN;
CREATE ROLE
test=# GRANT bookkeeper TO joe;
GRANT ROLE
```

这里的第一件事是创建了一个名为 bookkeeper 的角色。注意我们并不想让人们以 bookkeeper 登录，因此该角色被标记为 NOLOGIN。



还要注意如果使用 CREATE ROLE 子句，NOLOGIN 是默认值。如果更喜欢 CREATE USER 子句，其默认设置是 LOGIN。

然后，创建 joe 角色并且标记为 LOGIN。最后，bookkeeper 角色被指派给 joe 角色，这样他能做一个会计员被允许做的所有事情。

一旦用户就位，我们可以对已有的用户进行测试：

```
[hs@zenbook ~]$ psql test -U bookkeeper
psql: FATAL: role "bookkeeper" is not permitted to log in
```

不出所料，会计员角色不被允许登录系统。如果 joe 角色尝试会怎样？

```
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

这确实起到了预期的效果。不过，请注意命令提示已经发生了改变，这只是 PostgreSQL 用来向用户表示用户不是作为超级用户登录的方法。

一旦一个用户被创建，就可能需要对它进行修改。其中一个可能需要修改的是口令。在 PostgreSQL 中，允许用户修改自己的口令，可以这样做：

```
test=> ALTER ROLE joe PASSWORD 'abc';
ALTER ROLE
test=> SELECT current_user;
 current_user
--------------
 joe
(1 row)
```

ALTER ROLE 子句（或者 ALTER USER）将允许用户更改大部分在用户创建时设定的设置。不过，管理用户还有更多可做的事情。在很多情况下，我们想要为一个用户指派特殊的参数。使用 ALTER USER 子句就是一种方法：

```
ALTER ROLE { role_specification | ALL }
     [ IN DATABASE database_name ]
              SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL }
      [ IN DATABASE database_name ]
              SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL }
      [ IN DATABASE database_name ] RESET configuration_parameter
ALTER ROLE { role_specification | ALL }
      [ IN DATABASE database_name ] RESET ALL
```

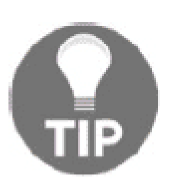
这种语法相当简单并且很直接。为了向读者展示这确实有用，笔者增加了一个真实世界的例子。假定 Joe 刚好生活在毛里求斯岛上。当他登录时，即便他的数据库服务器位于欧洲，他也希望设置为他所在的时区：

```
test=> ALTER ROLE joe SET TimeZone = 'UTC-4';
ALTER ROLE
test=> SELECT now();
              now
-------------------------------
 2017-01-09 20:36:48.571584+01
(1 row)

test=> q
 [hs@zenbook ~]$ psql test -U joe
...
test=> SELECT now();
              now
-------------------------------
```

```
 2017-01-09 23:36:53.357845+04
(1 row)
```

ALTER ROLE 子句将修改用户。只要 Joe 重新连接，就已经为他设置好了时区。



时区不会被立即改变。用户需要重新连接或者使用 SET ... TO DEFAULT 子句。

这里的重点是这种修改还可以用于一些内存参数，例如本书早前已经涵盖的 work_mem 等。

8.1.4　定义数据库级安全性

在实例级上配置好用户之后，现在可以深入数据库级，看看在数据库级可以做些什么。第一个出现的主要问题是，我们明确地允许 Joe 登录数据库实例，但是是谁或者是什么让 Joe 真正地连接到其中一个数据库？用户可能并不想让 Joe 访问系统中的所有数据库。限制对特定数据库的访问正是我们可以在这一个级别上实现的特性。

对于数据库，可以使用 GRANT 子句设置下列权限：

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
     | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

在数据库级有两种主要的权限值得密切关注。

- CREATE：允许某个人在数据库中创建方案，注意，CREATE 子句不允许创建表，它是有关方案的权限。在 PostgreSQL 中，表位于一个方案中，因此在能创建表之前必须先涉及方案级别。
- CONNECT：允许某个人连接到一个数据库。

现在的问题是，没有人显式地给 joe 角色指派 CONNECT 权限。那么那些权限到底从何而来？答案是：有一个名为 public 的东西，它和 Unix 世界中的 public 类似。如果在这个世界中大家都被允许做某事，那么 Joe 也被允许，他是公众的一部分。

重点是从能被删除和重命名这一点来看，public 并不是一个角色。读者可以简单地把它理解成等效于系统中的每一个人。

因此，为了确保不是每一个人都能在任何时间连接到任意数据库，可能不得不从公众收回 CONNECT。要做到这一点，可以作为超级用户连接并且修复这一问题：

```
[hs@zenbook ~]$ psql test -U postgres
...
```

```
test=# REVOKE ALL ON DATABASE test FROM public;
REVOKE
test=# q
[hs@zenbook ~]$ psql test -U joe
psql: FATAL: permission denied for database "test"
DETAIL: User does not have CONNECT privilege.
```

如你所见，joe 角色不再被允许连接。此时只有超级用户具有对 test 的访问。

通常，在其他数据库被创建之前就从 postgres 数据库中收回权限是一个不错的主意。这一概念背后的思想是那些权限将不再出现在那些新创建的数据库中。如果某人需要对一个特定数据库的访问，可以明确地授予权限，权限不再被自动授予。

如果想要允许 joe 角色连接到 test 数据库，可以作为超级用户尝试下面的命令：

```
[hs@zenbook ~]$ psql test -U postgres
      ...
test=# GRANT CONNECT ON DATABASE test TO bookkeeper;
GRANT
test=# q
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

基本上，这里有两种选择：

- 可以直接允许 joe 角色，这样只有 joe 角色将能连接。
- 可以授予权限给会计员角色。记住，joe 角色将从会计员角色继承所有权限，因此如果想让所有会计都能连接到这个数据库，对会计员角色分配权限好像是一种很有吸引力的想法。

如果对会计员角色授予权限，是不会有风险的，因为这个角色起初就不被允许登录实例，因此它纯粹是作为一个权限的来源而已。

8.1.5　调整方案级权限

一旦配置完了数据库级安全性，就有必要看看方案级的权限。

在实际进入这个级别前，笔者想要运行一个小测试：

```
test=> CREATE DATABASE test;
ERROR: permission denied to create database
test=> CREATE USER xy;
ERROR: permission denied to create role
```

```
test=> CREATE SCHEMA sales;
ERROR: permission denied for database test
```

如你所见，Joe 不太走运，除了允许连接到数据库，他基本上什么都做不了。

不过，有一个小例外，这会让很多人吃惊：

```
test=> CREATE TABLE t_broken (id int);
CREATE TABLE
test=> d
              List of relations
 Schema |        Name        |  Type | Owner
--------+------------------+-------+-------
 public | t_broken          | table | joe
(1 rows)
```

默认情况下，public 被允许使用 public 模式，并且总是如此。如果对保护数据库的安全特别感兴趣，应确保这一问题得到妥善处理。否则，普通用户将可能在 public 模式中放入各种表，最后整个系统设置都会遭殃。还要记住如果某人被允许创建对象，这个人就是它的拥有者。拥有关系意味着创建者自动拥有所有的权限（包括毁掉该对象）。

为了从 public 中拿走那些权限，可以作为超级用户运行下面的命令：

```
test=# REVOKE ALL ON SCHEMA public FROM public;
REVOKE
```

从现在开始，再没有人能在没有权限的情况下在 public 模式中放东西：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_data (id int);
ERROR: no schema has been selected to create in
LINE 1: CREATE TABLE t_data (id int);
```

如你所见，这个命令将会失败。这里的重点是用户看到的错误消息，PostgreSQL 不知道在哪里放这些表。默认情况下，它将尝试把表放入下列方案之一：

```
test=> SHOW search_path ;
   search path
-----------------
 "$user", public
(1 row)
```

由于没有一个名叫 joe 的方案，那么就不能把表放入其中，PostgreSQL 将尝试 public 模式。由于没有权限，PostgreSQL 将抱怨它不知道把表放在哪里。

如果表被明确地加上前缀，情况将立即改变：

```
test=> CREATE TABLE public.t_data (id int);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE public.t_data (id int);
```

在这种情况下，用户将得到一个意料之中的错误消息。PostgreSQL 否决了对 public 模式的访问。

下一个逻辑上的问题是：在模式级别上可以设置哪些权限以给予 joe 角色更多能力？

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

CREATE 意味着某人可以在一个模式中放入对象。USAGE 表示某人被允许进入该模式。注意进入模式并不意味着可以真正使用该模式中的东西——那些权限还没有被定义。说白了，这只表示该用户能看到这个方案的系统目录项而已。

为了允许 joe 角色访问他之前创建的表，将需要下列命令（以超级用户执行）：

```
test=# GRANT USAGE ON SCHEMA public TO bookkeeper;
GRANT
```

现在，joe 角色能够按照预期读取他的表：

```
[hs@zenbook ~]$ psql test -U joe
test=> SELECT count(*) FROM t_broken;
 count
-------
     0
(1 row)
```

joe 角色还能增加和修改行，因为他恰好是这个表的拥有者。不过，尽管他已经能做很多事情，joe 角色还不够强大。考虑下列语句：

```
test=> ALTER TABLE t_broken RENAME TO t_useful;
ERROR: permission denied for schema public
```

让我们更仔细地看看实际的错误消息。如你所见，该消息抱怨的是有关该模式上的权限，而不是表本身上的权限（记住，joe 角色拥有该表）。为了解决这个问题，就必须在模式级别而不是在表级别上进行处理。请作为超级用户运行下列命令：

```
test=# GRANT CREATE ON SCHEMA public TO bookkeeper;
GRANT
```

joe 角色现在可以把他的表的名称改成一个更有用的名称：

```
[hs@zenbook ~]$ psql test -U joe
test=> ALTER TABLE t_broken RENAME TO t_useful;
ALTER TABLE
```

记住如果使用 DDL，这就是必需的。在笔者作为 PostgreSQL 支持服务提供商的日常工作中，笔者已经见过了若干这一点导致的问题。

8.1.6　使用表

在处理完绑定地址、网络认证、用户、数据库和方案之后，最后我们来到了数据库级别。下面的片段展示了对于一个表可以设置哪些权限：

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE
              | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
       | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

下面逐个解释一下那些权限。

- SELECT：允许用户读表。
- INSERT：允许用户向表加入行（这还包括复制等——它不仅与 INSERT 子句有关）。注意如果用户被允许插入并不表示就自动被允许读取。要读取自己已经插入的数据，用户需要能执行 SELECT 和 INSERT 子句。
- UPDATE：修改表的内容。
- DELETE：被用来从表中移除行。
- TRUNCATE：允许用户使用 TRUNCATE 子句。注意，DELETE 和 TRUNCATE 子句是两种单独的权限，因为 TRUNCATE 子句将会锁表，而 DELETE 子句不会（即便没有 WHERE 条件）。
- REFERENCES：允许创建外键。必须在引用列和被引用列上都有这一特权，否则无法创建外键。
- TRIGGER：允许创建触发器。

GRANT 子句有一个好处是可以同时设置一个方案中所有表的权限。

它很大程度上简化了调整权限的过程。还可以使用 WITH GRANT OPTION 子句。其

思想在于确保普通用户可以把权限传递给其他用户，这种做法的优点是可以极大地降低管理员的工作量。想象一个为数百个用户提供访问的系统——管理所有这些人本身就是一项庞大的工作，因此管理员可以找一些人自行管理数据的一个子集。

8.1.7　处理列级安全性

在某些情况下，并非所有人都被允许看见所有的数据。想象一家银行，某些人可以看到有关一个银行账户的全部信息，而其他人可能只能看到数据的一个子集。在真实世界的情况中，某人可能被允许读取余额列或者默认可能看不到人们贷款的利率。另一个例子是人们被允许查看人员的概要信息但看不到他们的照片或某些其他隐私信息。现在的问题是：怎样使用列级安全性？

为了展示，笔者将对属于 joe 角色的已有表增加一列：

```
test=> ALTER TABLE t useful ADD COLUMN name text;
ALTER TABLE
```

现在这个表由两列组成。这个例子的目标是确保一个用户只能看到其中一列：

```
test=> \d t useful
   Table "public.t useful"
 Column |   Type   | Modifiers
--------+---------+-----------
 id     | integer |
 name   | text    |
```

让我们创建一个用户并且让它能访问包含例子表的方案：

```
test=# CREATE ROLE paul LOGIN;
CREATE ROLE
test=# GRANT CONNECT ON DATABASE test TO paul;
GRANT
test=# GRANT USAGE ON SCHEMA public TO paul;
GRANT
```

不要忘记把 CONNECT 权利给予新人，因为在本章稍早的地方，CONNECT 已经被从 public 收回。因此为了确保我们能够有机会访问该表，显式授权是绝对必要的。

可以把 SELECT 权限给 paul 角色：

```
test=# GRANT SELECT (id) ON t useful TO paul;
GRANT
```

基本上，这已经足够。已经可以作为用户 paul 连接到数据并且读取该列：

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT id FROM t_useful;
 id
----
(0 rows)
```

如果使用列级权限，有一件重要的事情需要记住：应该停止使用 SELECT *，因为它已经无法再用：

```
test=> SELECT * FROM t useful;
ERROR: permission denied for relation t_useful
```

*仍然表示所有列，但由于这里没有办法访问所有列，访问尝试将会立即导致报错。

8.1.8　配置默认特权

到目前为止，我们已经配置了很多东西。现在自然而然会产生的麻烦是，如果新表被加入系统会发生什么？逐个处理这些表并且设置正确的权限可能会非常痛苦并且有风险。如果这些事情自动发生会不会很好？这就是 ALTER DEFAULT PRIVILEGES 子句负责的事情。其思想是给用户一种选项，它能让 PostgreSQL 在对象一出现时自动设置想要的权限。这样就再不会发生忘记设置那些权限的情况。

下面的列表展示了该语法说明的第一部分：

```
test=# h ALTER DEFAULT PRIVILEGES
Command:     ALTER DEFAULT PRIVILEGES
Description: define default access privileges
Syntax:
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target role [, ...] ]
    [ IN SCHEMA schema name [, ...] ]
    abbreviated_grant_or_revoke

where abbreviated_grant_or_revoke is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE
     | TRUNCATE
     | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
   ON TABLES
```

```
   TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
...
```

基本上，该语法的效果类似于 GRANT 子句，因此比较容易和直观地使用。为了向读者展示如何使用它，笔者编了一个简单的例子。其想法是如果 joe 角色创建一个表，paul 角色将自动能使用它：

```
test=# ALTER DEFAULT PRIVILEGES FOR ROLE joe
     IN SCHEMA public
     GRANT ALL ON TABLES TO paul;
ALTER DEFAULT PRIVILEGES
```

现在作为 joe 角色连接并且创建一个表：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_user (id serial, name text, passwd text);
CREATE TABLE
```

作为 paul 角色连接将证明该表已经被指派了正确的权限集：

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT * FROM t_user;
 id | name | passwd
----+------+--------
(0 rows)
```

8.2　深入行级安全性——RLS

到目前为止，一个表总是被作为整体显示。当表包含一百万行时，可以从中检索一百万行。如果某人有权利读取一个表，该权利就是有关整个表的。在很多情况下，这就足够了。但实际经常会想让一个用户不能看到所有行。

考虑下面这个真实世界的例子，一个会计为很多人做会计工作。该表包含应该对所有人可见的税率，因为每个人必须支付相同的税率。但是对于实际的交易，用户可能希望确保每个人只被允许查看他或者她自己的交易。

人员 A 不应被允许看到人员 B 的数据。此外，可能还有必要允许一个部门的老板查看公司中属于他这一部分的数据。

行级安全性就是为这种目的设计的，它允许用户以快速简便的方式构建多租户系统。配置那些权限的方法是使用策略。这里的 CREATE POLICY 命令为用户提供了一种

方式来编写那些规则：

```
test=# h CREATE POLICY
Command:     CREATE POLICY
Description: define a new row level security policy for a table
Syntax:
CREATE POLICY name ON table name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role name | PUBLIC | CURRENT USER | SESSION USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

为了向读者展示如何编写一条策略，笔者首先作为超级用户登录并且创建一个包含若干项的表：

```
test=# CREATE TABLE t_person (gender text, name text);
CREATE TABLE
test=# INSERT INTO t person VALUES
     ('male', 'joe'), ('male', 'paul'), ('female', 'sarah'), (NULL, 'R2-
D2');
INSERT 0 4
```

然后为 joe 角色的访问授权：

```
test=# GRANT ALL ON t person TO joe;
GRANT
```

到目前为止所做的事情都很普通，并且 joe 角色将能够实际读取整个表，因为 RLS 还没有就位。但是如果为该表启用行级安全性会发生什么？

```
test=# ALTER TABLE t_person ENABLE ROW LEVEL SECURITY;
ALTER TABLE
```

默认就有一条否决全部的策略，因此 joe 角色实际将得到一个空表：

```
test=> SELECT * FROM t_person;
 gender | name
--------+------
(0 rows)
```

事实上，这条默认策略很有意义，因为用户会被强制要求明确地设置权限。

现在这个表已经处于行级安全性控制之下，可以开始编写策略了（作为超级用户）：

```
test=# CREATE POLICY joe pol 1
              ON t_person
```

```
               FOR SELECT TO joe
               USING (gender = 'male');
CREATE POLICY
```

作为 joe 角色登录并且选择所有数据，将只会返回两行：

```
test=> SELECT * FROM t person;
 gender | name
---------+------
 male    | joe
 male    | paul
(2 rows)
```

让我们以更细致的方式观察一下笔者刚才创建的策略。读者看到的第一件事情是一条策略实际是有名字的。它还被连接到一个表并且允许特定操作（这里是 SELECT 子句）。然后就是 USING 子句，它基本上定义了 joe 角色将被允许看到什么。因此 USING 子句是附加到每一个查询的一种强制过滤，它只选择用户被假定可以看到的行。

现在假定由于某种原因决定也允许 joe 角色看到机器人。有两种选项可以实现这一目的。第一种选项是简单地使用 ALTER POLICY 子句更改现有的策略：

```
test=> h ALTER POLICY
Command:     ALTER POLICY
Description: change the definition of a row level security policy
Syntax:
ALTER POLICY name ON table name RENAME TO new name

ALTER POLICY name ON table_name
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

第二种选项是如下例所示创建第二条策略：

```
test=# CREATE POLICY joe pol 2
               ON t person
               FOR SELECT TO joe
               USING (gender IS NULL);
CREATE POLICY
```

这里的妙处是那些策略会被使用 OR 条件连接起来。因此，PostgreSQL 现在将返回三行而不是两行：

```
test=> SELECT * FROM t_person;
 gender | name
--------+-------
 male   | joe
 male   | paul
        | R2-D2
(3 rows)
```

R2-D2 角色现在也被包括在结果中，因为它匹配第二条策略。

为了向读者展示 PostgreSQL 如何运行该查询，笔者决定把该查询的执行计划包括进来：

```
test=> explain SELECT * FROM t person;
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on t_person (cost=0.00..21.00 rows=9 width=64)
   Filter: ((gender IS NULL) OR (gender = 'male'::text))
(2 rows)
```

如你所见，所有的 USING 子句都会被作为强制过滤条件加入该查询中。

读者可能已经注意到在语法定义中有两类子句。

- USING：这个子句过滤已经存在的行。它与 SELECT 和 UPDATE 子句等相关。
- CHECK：这个子句过滤将要被创建的新行，因此它们与 INSERT 和 UPDATE 子句等相关。

这里是如果我们尝试插入一行将发生的事情：

```
test=> INSERT INTO t_person VALUES ('male', 'kaarel');
ERROR: new row violates row-level security policy for table "t_person"
```

因为没有针对 INSERT 子句的策略，该语句自然会报错。下面是允许插入的策略：

```
test=# CREATE POLICY joe_pol_3
              ON t_person
              FOR INSERT TO joe
              WITH CHECK (gender IN ('male', 'female'));
CREATE POLICY
```

如下面的列表所示，joe 角色被允许向表中增加男性和女性：

```
test=> INSERT INTO t_person VALUES ('female', 'maria');
INSERT 0 1
```

不过，还有一个例外，考虑下面的例子：

```
test=> INSERT INTO t_person VALUES ('female', 'maria') RETURNING *;
ERROR: new row violates row-level security policy for table "t_person"
```

回忆一下，现在只有一条选择男性的策略。这里的麻烦在于该语句将返回一位女性，这是不被允许的，因为 joe 角色受到一条只选择男性的策略约束。

RETURNING *子句只有对男性才会实际有效：

```
test=> INSERT INTO t person VALUES ('male', 'max') RETURNING *;
 gender | name
--------+------
 male   | max
(1 row)

INSERT 0 1
```

如果用户不想要这种行为，必须编写一条包含正确 USING 子句的策略。

8.3　检查权限

当所有权限被设置好时，有时候有必要知道谁有什么权限。对于管理员来说找出谁被允许做什么是至关重要的。不幸的是，这一过程并不容易并且要求一些知识。通常，笔者是一个命令行的超级粉丝。但是，在查看权限系统时，实在有必要使用一种图形化用户接口。

在向读者展示如何读取 PostgreSQL 权限之前，笔者将为 joe 角色指派权利，这样可以在下一步中观察：

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

检索有关权限的信息可以用 psql 中的 z 命令完成：

```
test=# x
Expanded display is on.
test=# z t_person
Access privileges
-[ RECORD 1 ]-----+-----------------------------------------------
----
Schema            | public
Name              | t person
Type              | table
Access privileges | postgres=arwdDxt/postgres
```

```
+
                      | joe=arwdDxt/postgres
Column privileges |
Policies              | joe_pol_1 (r):
+
                      | (u): (gender = 'male'::text)
+
                      | to: joe
+
                      | joe_pol_2 (r):
+
                      | (u): (gender IS NULL)
+
                      | to: joe
+
                      | joe_pol_3 (a):
+
                      | (c): (gender = ANY (ARRAY['male'::text,
'female'::text]))+
                      | to: joe
```

它将返回所有那些策略以及有关 Access privileges 的信息。不幸的是，那些信息不容易阅读并且笔者感觉它们并没有被管理员们广泛地理解。在我们的例子中，joe 角色从 postgres 得到了 arwdDxt。那些缩写到底意味着什么？

- a：追加，用于 INSERT 子句。
- r：读取，用于 SELECT 子句。
- w：写入，用于 UPDATE 子句。
- d：删除，用于 DELETE 子句。
- D：被用于 TRUNCATE 子句（当它被引入时，t 已经被用掉了）。
- x：被用于引用。
- t：被用于触发器。

如果用户不了解那些缩写，还有第二种方法来改善可读性。考虑下列函数调用：

```
test=# SELECT * FROM aclexplode('{joe=arwdDxt/postgres}');
 grantor | grantee | privilege_type | is_grantable
---------+---------+----------------+--------------
      10 |   18481 | INSERT         | f
      10 |   18481 | SELECT         | f
      10 |   18481 | UPDATE         | f
      10 |   18481 | DELETE         | f
```

```
      10 |    18481 | TRUNCATE       | f
      10 |    18481 | REFERENCES     | f
      10 |    18481 | TRIGGER        | f
(7 rows)
```

如你所见，权限集合被作为一个简单表返回，就会让阅读变得更加容易。

8.4　再分配对象和删除用户

在指派权限和限制访问之后，可能会发生用户将被从系统中删除的情况。不出所料，做这件事的命令是 DROP ROLE 和 DROP USER：

```
test=# h DROP ROLE
Command:     DROP ROLE
Description: remove a database role
Syntax:
DROP ROLE [ IF EXISTS ] name [, ...]
```

让我们试一下：

```
test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects
     depend on it
DETAIL: target of policy joe pol 3 on table t person
target of policy joe pol 2 on table t person
target of policy joe pol 1 on table t person
privileges for table t_person
owner of table t_user
owner of sequence t_user_id_seq
owner of default privileges on new relations belonging to role joe
     in schema public
owner of table t_useful
```

PostgreSQL 会发出错误消息，因为一个用户只有在被剥夺了所有东西以后才能被移除。其原因是，假定某人拥有一个表，PostgreSQL 应该怎么处理那个表？必须要有人拥有它们。为了把表从一个用户重新分配到下一个用户，考虑一下 REASSIGN 子句：

```
test=# h REASSIGN
Command:     REASSIGN OWNED
Description: change the ownership of database objects owned
     by a database role
Syntax:
```

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
               TO { new_role | CURRENT_USER | SESSION_USER }
```

该语法也非常简单并且有助于简化这种交接处理。这里有一个例子：

```
test=# REASSIGN OWNED BY joe TO postgres;
REASSIGN OWNED
```

因此让我们再次尝试删除 joe 角色：

```
test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects depend on it
DETAIL: target of policy joe_pol_3 on table t_person
target of policy joe_pol_2 on table t_person
target of policy joe_pol_1 on table t_person
privileges for table t_person
owner of default privileges on new relations belonging to role joe
     in schema public
```

如你所见，问题列表已经被显著地缩短。现在我们能够做的是逐个解决这些问题并且删除该角色。笔者没有发现有捷径可走，唯一可以让这种处理更加有效的方法是确保尽可能少地直接把权限分配给用户。尽量尝试用角色来抽象权限，然后让很多人都使用角色。即便个别权限被直接分配给用户，通常事情也会比较容易处理。

8.5　总　　结

数据库安全性是一个非常广的领域，本章十来页的篇幅很难覆盖 PostgreSQL 安全性的所有方面。很多诸如 SELinux、安全性定义者/调用者之类的内容都没有被触及。不过，在本章中读者学到了作为一个 PostgreSQL 开发者和 DBA 将会面对的最常见的问题。读者学到了如何避免基本的陷阱以及如何让系统更加安全。

在第 9 章中，读者将学习有关 PostgreSQL 流复制和增量备份的内容。该章还将涵盖故障转移场景的相关内容。

第 9 章　处理备份和恢复

在本书的第 8 章，笔者尝试让读者了解了以最简单和最有益的方法保护 PostgreSQL 的相关知识。本章的主题将是备份和恢复。执行备份应当是一种定期任务，并且每一个管理员都应该关注这一关键性的活动。幸运的是，PostgreSQL 提供了创建备份的便利方法。

本章将涵盖下列主题：

- 运行 pg_dump。
- 部分转储数据。
- 还原备份。
- 利用并行性。
- 保存全局数据。

在本章结束时，读者将能够设置正确的备份机制。

9.1　执行简单转储

如果用户已经在运行一个 PostgreSQL，基本上有两种主要的方法来执行备份：

- 逻辑备份（抽取一个 SQL 脚本来表示数据）。
- 事务日志传送。

事务日志传送的思想是归档对数据库的二进制改变。大部分人声称事务日志传送是执行备份的唯一方式，但在笔者看来并不一定是那样。

很多人依靠 pg_dump 简单地提取数据的文本表示。pg_dump 也是最古老的创建备份的方法，并且从 PostgreSQL 项目开始至今都在被使用（事务日志传送很久以后才被加入）。每一个 PostgreSQL 管理员迟早将接触到 pg_dump，因此有必要了解它究竟怎么工作以及它能做些什么。

9.1.1　运行 pg_dump

我们想要做的第一件事是创建一个简单的文本形式的转储：

```
[hs@linuxpc ~]$ pg_dump test > /tmp/dump.sql
```

这是能想象到的最简单的备份。pg_dump 登录本地数据库实例，连接到数据库 test 并且开始抽取所有的数据，这些数据将被发送到 stdout，并且被重定向到文件。这样做的好

处是标准输出可以把 Unix 系统的所有灵活性都发挥出来。用户可以很容易地使用一个管道来压缩数据或者做任何想做的事情。

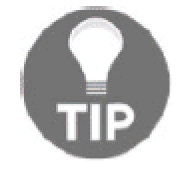
在很多情况下，使用者可能想要作为一个不同的用户运行 pg_dump。所有 PostgreSQL 客户端程序都支持一套一致的命令行参数来传递用户信息。如果想要设置用户，可以使用-U 标志：

```
[hs@linuxpc ~]$ pg_dump -U whatever_powerful_user test > /tmp/dump.sql
```

下列参数集合可以在所有 PostgreSQL 客户端程序中找到：

```
...
Connection options:
 -d, --dbname=DBNAME          database to dump
 -h, --host=HOSTNAME          database server host or
                              socket directory
 -p, --port=PORT              database server port number
 -U, --username=NAME          connect as specified database user
 -w, --no-password            never prompt for password
 -W, --password               force password prompt (should
                              happen automatically)
   --role=ROLENAME            do SET ROLE before dump
...
```

只要把想要的信息传递给 pg_dump 并且具有足够的权限，PostgreSQL 将会取得数据。这里的重点是看看该程序实际如何工作。简单来说，pg_dump 会连接到数据库并且打开一个大型的可重复读事务，该事务会简单地读取所有的数据。记住，可重复读确保 PostgreSQL 创建数据的一份一致的快照，它在事务中至始至终都不会改变。换句话说，转储总是一致的——不会有外键被违背。pg_dump 的输出是转储开始时数据的一份快照。这里一致性是一个关键因素，它还表示转储运行时对数据的更改将不会出现在备份中。



转储简单地读取所有东西——因此，没有单独的转储权限，只要能读对象，就能备份它。

还要注意备份的默认格式是文本格式。这意味着可以安全地从 Solaris 抽取的数据，移动到某种其他 CPU 架构上。在二进制备份的情况下，这显然不可能，因为磁盘格式依赖于 CPU 架构。

9.1.2　传递口令和连接信息

如果仔细观察 9.1.1 节中展示的连接参数，读者将会注意到没有办法向 pg_dump 传递

口令。用户可以使用口令提示来要求口令，但是没有办法用命令行选项把口令传递给 pg_dump。其原因很简单：口令可能会出现在进程表中并且可能因此对其他人可见，所以不能支持这种方式。现在的问题是：如果服务器上的 pg_hba.conf 强制要求口令，客户端程序怎样才能提供？

有几种方式可以做到：

- 利用环境变量。
- 利用.pgpass。
- 使用服务文件。

在本节中，读者将学到所有 3 种方法。

1. 使用环境变量

一种方法是使用环境变量传递各种参数。如果信息没有被明确地传递给 pg_dump，它将在预定义的环境变量中查找缺少的信息。所有可能的环境变量的列表可以在下面的网址找到：https://www.postgresql.org/docs/9.6/static/libpq-envars.html。

下面展示了备份通常需要的一些环境变量。

- PGHOST：告诉系统要连接哪台主机。
- PGPORT：定义要被使用的 TCP 端口。
- PGUSER：告诉客户端程序要使用的用户。
- PGPASSWORD：包含要使用的口令。
- PGDATABASE：是要连接的数据库名称。

使用这些环境变量的好处是口令不会出现在进程表中。不过，还不仅限于此，考虑下面的例子：

```
psql -U ... -h ... -p ... -d ...
```

假定读者是一个系统管理员：你真想每天都多次输入这么长的行吗？如果每次都是用相同的主机，只需要设置那些环境变量并且用 psql 连接就行：

```
[hs@linuxpc ~]$ export PGHOST=localhost
[hs@linuxpc ~]$ export PGUSER=hs
[hs@linuxpc ~]$ export PGPASSWORD=abc
[hs@linuxpc ~]$ export PGPORT=5432
[hs@linuxpc ~]$ export PGDATABASE=test
[hs@linuxpc ~]$ psql
psql (9.6.1)
Type "help" for help.
```

如你所见，这里不再有命令行参数。只是输入 psql 就可以登入。



所有基于标准 C 库（libpq）的应用都能理解那些环境变量，因此可以把它们用于 psql 和 pg_dump 之外的很多其他应用。

2. 利用.pgpass

一种很常见的存放登录信息的方法是使用.pgpass 文件。其思想很简单：在用户的主目录下放一个名为.pgpass 的文件，并且将用户的登录信息放在其中。格式也很简单：

```
hostname:port:database:username:password
```

下面是一个例子：

```
192.168.0.45:5432:mydb:xy:abc
```

PostgreSQL 提供了一种很好的额外功能：大部分域都可以包含*。例如：

```
*:*:*:xy:abc
```

这意味着在每一台主机上、在每个端口上、对每个数据库，名为 xy 的用户都将使用 abc 作为口令。要让 PostgreSQL 使用.pgpass 文件，确保设置好正确的文件权限：

```
chmod 0600 ~/.pgpass
```

.pgpass 也可以被用在 Windows 系统上。在这种情况下，该文件可以在%APPDATA%\postgresql\pgpass.conf 路径上找到

3. 使用服务文件

不过，还不只可以使用.pgpass 文件，用户还可以利用服务文件。方法如下，如果用户想要多次连接到相同的服务器，可以创建一个.pg_service.conf 文件，其中将保存所有需要的连接信息。

这里是一个.pg_service.conf 文件的例子：

```
Mac:~ hs$ cat .pg service.conf

# a sample service
[hansservice]
host=localhost
port=5432
dbname=test
user=hs
```

```
password=abc

[paulservice]
host=192.168.0.45
port=5432
dbname=xyz
user=paul
password=cde
```

要连接到其中一个服务，只需要设置环境并且连接：

```
iMac:~ hs$ export PGSERVICE=hansservice
```

现在可以无须向 psql 传递口令而建立连接：

```
iMac:~ hs$ psql
psql (9.6.1)
Type "help" for help.
test=#
```

或者，可以这样做：

```
psql service=hansservice
```

9.1.3　提取数据的子集

到目前为止，读者已经看到了如何转储一整个数据库。但是，用户可能并不希望这样做。在很多情况下，用户可能只是想抽取表或者方案的一个子集。

pg_dump 可以做到这一点并且提供了数个开关。

- -a：只转储数据而不转储数据结构。
- -s：只转储数据结构而跳过数据。
- -n：只转储一个特定的方案。
- -N：转储所有东西但排除特定的方案。
- -t：只转储特定的表。
- -T：转储所有东西但排除特定的表（如果想排除日志表等就会用到）。

部分转储有助于大幅度提高速度。

9.1.4　处理多种数据格式

到目前为止，读者已经看到 pg_dump 可以被用来创建文本文件。但问题是文本文件只能被完全重放，如果用户已经保存了一个完整的数据库，用户只能重放整个数据库。

在很多情况下，这并非用户想要的。因此，PostgreSQL 还有一些额外的格式提供更多的功能。

目前，PostgreSQL 支持 4 种格式：

```
-F, --format=c|d|t|p         output file format
                             (custom, directory, tar,
                             plain text (default))
```

读者已经看过了纯文本格式，它就是普通的文本。此外，用户可以使用自定义格式。自定义格式是一种压缩过的转储，其中包括一个表的内容。这里有两种方法创建一个自定义格式的转储：

```
[hs@linuxpc ~]$ pg dump -Fc test > /tmp/dump.fc
[hs@linuxpc ~]$ pg_dump -Fc test -f /tmp/dump.fc
```

除了表内容，压缩转储还有一个优点：它更小。经验是一个自定义格式的转储大约比要备份的数据库实例小 90%。当然，这高度依赖于索引的数量，但对于很多数据库应用来说这一估计总是对的。

一旦创建了备份，就可以观察备份文件：

```
[hs@linuxpc ~]$ pg_restore --list /tmp/dump.fc
;
; Archive created at 2017-01-04 15:44:56 CET
;     dbname: test
;     TOC Entries: 18
;     Compression: -1
;     Dump Version: 1.12-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 9.6.1
;     Dumped by pg dump version: 9.6.1
;
; Selected TOC Entries:
;
3103; 1262 16384 DATABASE - test hs
3; 2615 2200 SCHEMA - public hs
3104; 0 0 COMMENT - SCHEMA public hs
1; 3079 13350 EXTENSION - plpgsql
3105; 0 0 COMMENT - EXTENSION plpgsql
187; 1259 16391 TABLE public t test hs
...
```

pg_restore --list 将会返回该备份的内容表。

使用自定义格式已经是一种很好的想法，因为备份的尺寸将会缩小。但是，还不仅如此，-Fd 命令将以目录格式创建备份。用户现在得到一个包含若干文件的目录而不是单个文件：

```
[hs@linuxpc ~]$ mkdir /tmp/backup
[hs@linuxpc ~]$ pg dump -Fd test -f /tmp/backup/
[hs@linuxpc ~]$ cd /tmp/backup/
[hs@linuxpc backup]$ ls -lh
total 86M
-rw-rw-r--. 1 hs hs  85M Jan 4 15:54 3095.dat.gz
-rw-rw-r--. 1 hs hs  107 Jan 4 15:54 3096.dat.gz
-rw-rw-r--. 1 hs hs 740K Jan 4 15:54 3097.dat.gz
-rw-rw-r--. 1 hs hs   39 Jan 4 15:54 3098.dat.gz
-rw-rw-r--. 1 hs hs 4.3K Jan 4 15:54 toc.dat
```

目录格式的一个优点是可以使用多个 CPU 核来执行备份。在纯文本格式或自定义格式的情况下，pg_dump 将仅使用一个进程，而目录格式改变了这一规则。下面的例子展示了用户如何能告诉 pg_dump 使用 4 个核（任务）：

```
[hs@linuxpc backup]$ rm -rf *
[hs@linuxpc backup]$ pg_dump -Fd test -f /tmp/backup/ -j 4
```

注意在数据库中有越多对象，就会有越大的潜在提速空间。

9.2 重放备份

只有备份是无意义的，备份存在的意义就是为了有朝一日用它来进行重放。幸运地是，重放备份很容易。如果已经创建一个纯文本备份，可以简单地拿到那个 SQL 文件并且执行它：

```
psql your_db < your_file.sql
```



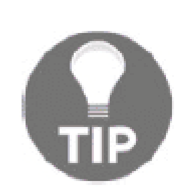
纯文本备份就是一个包含所有东西的文本文件。用户总是可以简单地重放文本文件。

如果在备份时使用了自定义格式或者目录格式，用户可以使用 pg_restore 来重放备份。pg_restore 允许用户做所有奇妙的事情，例如只重放一个数据库的一部分等。不过在大部分情况下，用户将会简单地重放整个数据库。在笔者的例子中，笔者将创建一个空数据库并且重放一个自定义格式的转储：

```
[hs@linuxpc backup]$ createdb new_db
[hs@linuxpc backup]$ pg_restore -d new_db -j 4 /tmp/dump.fc
```

注意，pg_restore 将把数据增加到一个现有的数据库。如果数据库非空，pg_restore 可能会出错但是仍会继续。

-j 再一次被用来使用多个进程。在笔者的例子中，4 个核被用来重放数据（这只对重放多个表有效）。



如果使用目录格式，用户可以简单地传入目录的名字而不是文件的名字。

就性能而言，如果处理的是少量或者中等量的数据，转储是一种好的方案。它有两个主要缺点：

- 用户将得到一个快照，因此最后一个快照之后的所有东西都将被丢失。
- 从零重建一个转储相比二进制备份来说更慢，因为所有的索引都必须被重建。

因此，我们将在第 10 章中介绍二进制备份。

9.3　处理全局数据

在 9.2 节中，读者已经学到了 pg_dump 和 pg_restore，它们两个是创建备份时的关键程序。最重要的是，pg_dump 创建数据库转储——它工作在数据库级别上。如果想要备份整个实例，必须使用 pg_dumpall 或者单独转储所有数据库。在进一步深入之前，有必要看看 pg_dumpall 如何工作：

```
pg_dumpall > /tmp/all.sql
```

pg_dumpall 将逐个连接到数据库并且把备份结果发送到标准输出，这里可以用 Unix 处理输出。pg_dumpall 可以像 pg_dump 一样被使用。不过，它有一些缺点。它不支持自定义或者目录格式，因此不能提供多核支持——用户将受制于单线程。

不过，pg_dumpall 也不是只有缺点。记住用户是存在于实例级别上。如果用户创建一个普通的数据库转储，用户将得到所有的权限但不会得到所有的 CREATE USER 语句。那些全局数据不会被包括在普通转储中——它们只会被 pg_dumpall 抽取出来。

如果用户只想要全局数据，用户可以使用-g 选项运行 pg_dumpall：

```
pg_dumpall -g > /tmp/globals.sql
```

在大部分情况下，用户可能想要运行“pg_dumpall –g”以及自定义或目录格式转储来抽取其实例。一个简单的备份脚本看起来像这样：

```
#!/bin/sh

BACKUP DIR=/tmp/

pg dumpall -g > $BACKUP DIR/globals.sql

for x in $(psql -c "SELECT datname FROM pg database
           WHERE datname NOT IN ('postgres', 'template0', 'template1')"
postgres -A -t)
do
  pg_dump -Fc $x > $BACKUP_DIR/$x.fc
done
```

它将首先转储全局数据，然后在一个数据库列表上循环将它们逐个抽取为自定义格式。

9.4　总　　结

在本章中，读者从大体上学到了创建备份和转储。到目前为止，二进制备份还没有被涉及，但读者已经能够从服务器抽取文本备份以便以最简单的方式保存和重放其数据。

第 10 章将涉及事务日志传送、流复制以及二进制备份。读者将学到如何使用 PostgreSQL 自带的工具来复制实例。

第 10 章　理解备份和复制

在本书的第 9 章中，读者学到了很多有关备份和恢复的内容，这些知识对于管理来说非常重要。到目前，本书只涉及了逻辑备份，笔者将在本章中改变这一点。

本章的内容全都与 PostgreSQL 的事务日志有关，用户可以用它来改进设置并且让系统更加安全。

本章将覆盖下列主题：

- 事务日志可以做什么以及为何需要它。
- 执行时间点恢复。
- 设置流复制。
- 复制冲突。
- 监控复制。
- 同步复制 vs. 异步复制。
- 理解时间线。

在本章结束时，读者将能够设置事务日志归档和复制。但要记住这一点：本章绝对不是一份复制特性的全面指南，它只算是一份简介而已。完全覆盖复制这一主题可能需要大约 500 页的篇幅。作为对比，Packt 单独出版的《PostgreSQL Replication》一书就接近 400 页。

本章将以更紧凑的形式覆盖最基本的内容。

10.1　理解事务日志

每一种现代数据库系统都提供功能来确保系统能够从崩溃（假如某物出问题或者某人拔掉了插头）中幸免，而且对于文件系统和数据库系统都应能得到这样的保证。

PostgreSQL 还提供了一种方法确保崩溃不会损伤数据完整性或者数据本身。它能保证在电源被切断的情况下，系统将总是能够重新回到正常状态并且做自己的工作。

提供这种安全性的方法被称为预写式日志（WAL）或者 xlog。其思想是不直接写入数据文件，而是先写入日志。为什么要这样做呢？想象一下用户正在写某些数据：

```
INSERT INTO data ... VALUES ('12345678');
```

假定数据被直接写入数据文件。如果操作在中间某处失败，数据文件将会被损坏。

它可能会包含写入了一半的行、没有索引指针的列、缺少提交信息等。由于硬件并不能真正保障大块数据的原子写，必须找到一种方法来让写入更加鲁棒。通过写入日志而不是直接写入文件就能解决这一问题。



在 PostgreSQL 中，事务日志由记录构成。

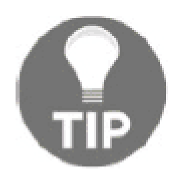
单次的写操作可能由数个日志记录组成，它们都有校验码并且被链接在一起。单个事务可能包含 B-树、存储管理器、提交记录以及更多种类的记录。每一类对象都有自己的 WAL 项并且确保该对象能够在崩溃时幸免。如果出现崩溃，PostgreSQL 启动时将基于事务日志修复数据文件以确保不会有永久性损伤发生。

10.1.1　察看事务日志

在 PostgreSQL 中，WAL 通常可以在 data 目录（除非在 initdb 时指定其他目录）的 pg_xlog 目录中找到。下面是其形式：

```
[postgres@zenbook pg_xlog]$ pwd
/var/lib/pgsql/9.6/data/pg_xlog
[postgres@zenbook pg xlog]$ ls -l
total 688132
-rw-------. 1 postgres postgres 16777216 Jan 19 07:58
0000000100000000000000CD
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04
0000000100000000000000CE
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04
0000000100000000000000CF
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04
0000000100000000000000D0
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04
0000000100000000000000D1
-rw-------. 1 postgres postgres 16777216 Jan 13 17:04
0000000100000000000000D2
```

在其中可以看到事务日志总是 16MB 的文件，名称由 24 个数字构成。文件名的编号是十六进制，如你所见，CF 后面就是 D0。文件总是固定尺寸。



在 PostgreSQL 中，事务日志文件的数量与事务的尺寸无关。我们可以用很小的事务日志文件集合轻易地运行数 TB 的事务。

10.1.2　理解检查点

正如笔者前面已经提到的，每一次更改都被以二进制格式（不包含 SQL）写入 WAL。这会导致一个问题：数据库服务器无法永远保持对 WAL 的写入，因为随着时间流逝它会消耗越来越多的空间。因此在某个时间点事务日志必须被重复利用，这个时间点由后台自动发生的**检查点（checkpoint）**确定。其思想如下：当数据被写入时，它首先进入事务日志，然后脏缓冲区被放入共享缓冲区中。那些脏缓冲区必须进入磁盘并且会被后台写入器或者检查点写出到数据文件中。只要到目前为止的所有脏缓冲区都已经被写出，事务日志就能被删除。



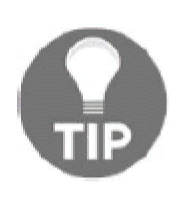
绝对不要手工删除事务日志文件。如果那样做了，在出现崩溃事件时，数据库服务器将不能重新启动，并且被删除的日志文件的磁盘空间将被回收用于新进入的事务。一定不要手工去触碰事务日志，PostgreSQL 会自行照顾好它们，人工对它们做任何事情实际上都是有害的。

10.1.3　优化事务日志

检查点会自动由服务器触发。不过，有一些配置参数的设置决定何时发动检查点。postgresql.conf 文件中的下列参数负责处理检查点：

```
#checkpoint_timeout = 5min          # range 30s-1d
#max_wal_size = 1GB
#min_wal_size = 80MB
```

有两种原因会发起检查点，时间用完或者空间用完。两次检查点之间的最大时间由 checkpoint_timeout 变量定义。为存储事务日志提供的空间大小将在 min_wal_size 和 max_wal_size 变量之间变化。PostgreSQL 将以一种方法自动地触发检查点，该方法会让实际需要的空间位于上述的两个变量值之间。



max_wal_size 变量是一个软限制，PostgreSQL（在重负载下）可以临时需要更多一点空间。换句话说，如果事务日志位于一个单独的磁盘上，有必要确保实际有更多的一点空间可以用来存储 WAL。

用户在 PostgreSQL 9.6 和 10.0 中如何调节事务日志？在 9.6 中，已经对后台写入器和检查点机制做出了一些改变。在更老的版本中，在某些用例下较小的检查点距离从性能的角度来说是有意义的。在 9.6 及其后的版本中，这种情况已经大有改观，较大的检查点距离基本上总是最有利的，因为那样有很多优化可以被应用在数据库和 OS 级别上以提高

速度。最值得一提的优化是在块被写出前先会被排序，这可以很大程度上降低机械磁盘的随机 I/O。

不仅如此，大的检查点距离实际上会减少被创建的 WAL 量。是的，就是这样——较大的检查点距离导致较少的 WAL。

其原因很简单。只要一个块在一个检查点后第一次被修改，它就会被完全发送到 WAL。如果这个块经常被更改，只有更改会被放入日志中。较大的距离说穿了会导致较少的全页写，这进而会减少起初创建的 WAL 量。如笔者的一篇博文中所述，区别可能会非常大：http://www.cybertec.at/checkpoint-distance-and-amount-of-wal/。

不仅如此，PostgreSQL 还允许我们配置检查点应该是短暂而猛烈还是应该散布在一段较长的时间段上。默认值是 0.5，这表示检查点处理应该在当前检查点和下一个检查点的正中间就能够完成：

```
#checkpoint_completion_target = 0.5
```

增大这个值基本上表示检查点被拉长并且不那么猛烈。在很多情况下已经证实，较大的值有利于拉平因为猛烈执行检查点导致的 I/O 激增。

10.2　事务日志归档和恢复

在对事务日志简单介绍之后，现在让我们把注意力放在事务日志归档处理上。如你所见，事务日志包含对存储系统所做的二进制更改的序列。因此，为什么不用它来复制数据库实例并且做很多其他很酷的事情呢？

10.2.1　为归档进行配置

我们想在本章中实现的第一件事情是创建一个配置来执行标准的时间点恢复（PITR）。PITR 相比普通转储有若干优势：

- 将丢失较少的数据，因为可以恢复一个特定的时间点而不只是固定的备份点。
- 恢复将会更快，因为索引不需要从头创建。它们只是被复制过来就可以投入使用。

PITR 的配置很容易，只需要在 postgresql.conf 文件中做少量设置：

```
wal_level = replica     # used to be "hot standby" in older versions
max_wal_senders = 5     # at least 2, better at least 2
```

wal_level 变量说明服务器被假定为产生足够多的事务日志以允许 PITR。如果 wal_level 变量被设置为 minimal（直到 PostgreSQL 9.6 都是默认值），事务日志将只包含

恢复单节点设置所需的信息——这些信息不足以处理复制。从最新的补丁看来，在 PostgreSQL 10.0 中那些默认值都已经被更改，因此设置归档甚至会更加容易。

max_wal_senders 变量将允许我们从服务器流式传送 WAL。它将允许用户使用 pg_basebackup 而不是传统基于文件的备份来创建一个初始备份，其优势是 pg_basebackup 更加容易使用。

WAL 流的思想是把创建的事务日志复制到某个可以安全恢复它的地方。基本上，有两种方法来传送 WAL：

- 使用 pg_receivexlog。
- 使用文件系统方式归档。

在本节中，读者将看到如何设置第二种选项。在普通操作期间，PostgreSQL 保持对那些 WAL 文件的写入。当 postgresql.conf 文件中 archive_mode = on 时，PostgreSQL 将为每一个文件调用 archive_command 变量。

其配置可能看起来是这样——首先，可以创建一个存储那些事务日志文件的目录：

```
mkdir /archive
chown postgres.postgres archive
```

下面的项可以在 postgresql.conf 文件中更改：

```
archive mode = on
archive_command = 'cp %p /archive/%f'
```

重启后将启用归档，但让我们先配置 pg_hba.conf 文件以最小化停机时间。

注意用户可以把任何命令放入 archive_command 变量中。很多人使用 rsync、scp 等把他们的 WAL 文件传送到安全的位置。如果用户的脚本返回 0，PostgreSQL 将假定该文件已经被归档。如果返回其他任何值，PostgreSQL 将尝试重新归档该文件。这是必要的，因为数据库引擎必须确保没有文件被丢失。如要执行恢复处理，就不允许失去任何一个文件。

10.2.2　配置 pg_hba.conf 文件

现在 postgresql.conf 文件已经被成功地配置好，接着需要为流复制配置 pg_hba.conf 文件。注意，只有计划使用 pg_basebackup 才有必要这样做，它是当前最好的基础备份创建工具。

基本上，pg_hba.conf 文件中的选项和已经在第 8 章中见过的一样。只有一点需要记住：

```
# Allow replication connections from localhost, by a user with the
# replication privilege.
```

```
local      replication     postgres                          trust
host       replication     postgres          127.0.0.1/32    trust
host       replication     postgres          ::1/128         trust
```

用户可以定义标准的 pg_hba.conf 文件规则，重点是第二列提到了 replication。普通规则是不够的——确实有必要增加明确的复制权限。还要记住不需要作为一个超级用户来执行复制，可以创建一个特定的只被允许登录和复制的用户。

现在 pg_hba.conf 文件已经被正确地配置，PostgreSQL 可以被重启了。

10.2.3　创建基础备份

在告诉了 PostgreSQL 要归档 WAL 文件之后，现在是时候创建第一个备份了。其思想是得到一个备份并且基于这个备份重放 WAL 文件以达到任意时间点。

要创建初始备份，用户可以求助于 pg_basebackup，它是一个用来执行备份的命令行工具。让我们调用 pg_basebackup 并且看看它如何工作：

```
pg_basebackup -D /some_target_dir
         -h localhost
         --checkpoint=fast
         --xlog-method=stream
```

如你所见，笔者这里使用了 4 个参数。

- -D：用户想让这个基础备份放在哪里？PostgreSQL 要求一个空目录，在备份结束时，用户将在这里（目标）看到服务器数据目录的一份备份。
- -h：指示主服务器（来源）的 IP 地址或者名称。这是想要备份的服务器。
- --checkpoint=fast：通常 pg_basebackup 会等待主服务器到检查点。其原因是重放处理必须从某处开始，而一个检查点能保证到某个特定点的数据都已经被写入，因此 PostgreSQL 能够安全地跳到那里并且开始重放处理。说穿了，也可以不用--checkpoint=fast 参数。但是，那种情况下 pg_basebackup 可能会花一些时间才开始复制数据。检查点可能会相距一小时，这可能会不必要地延迟备份。
- --xlog-method=stream：默认情况下，pg_basebackup 连接到主服务器并且开始复制文件过来。现在，要记住那些文件在被复制时还会被修改，因此到达备份中的数据是不一致的。这种不一致性可以在恢复处理期间使用 WAL 修复。不过，备份本身确实是不一致的。通过增加--xlogmethod=stream 参数，就可以创建一个自包含的备份。它可以被直接启动而无须重放事务日志（如果只是想克隆一个实例且不使用 PITR，这种方法就很不错）。

1. 限制备份的带宽

当 pg_basebackup 开始时，它会尝试尽快完成其工作。如果用户有一个好的网络连接，pg_basebackup 无疑可以从远程服务器每秒取得数百兆字节。如果用户服务器的 I/O 系统较弱，这就意味着 pg_basebackup 可能会很容易地吃掉所有的资源，最终用户可能会体验到不好的性能，因为他们的 I/O 请求太慢。

为了控制最大传输率，pg_basebackup 提供了下面的选项：

```
 -r, --max-rate=RATE maximum transfer rate to transfer data directory
(in kB/s, or use suffix "k" or "M")
```

在创建基础备份时，要确保主服务器上的磁盘系统实际能够承受这种负载。因此，调整传输率就变得很有意义了。

2. 映射表空间

如果在目标系统上使用的是相同的文件系统布局，那么可以直接调用 pg_basebackup。如果不是这样，pg_basebackup 允许把主服务器文件系统布局映射为想要的布局：

```
-T, --tablespace-mapping=OLDDIR=NEWDIR
              relocate tablespace in OLDDIR to NEWDIR
```



如果系统很小，把所有东西都放在一个表空间中是不错的做法。

只要 I/O 不是问题（可能因为只需要管理若干 GB 的数据）都可以这样做。

3. 使用不同的格式

pg_basebackup 可以创建多种格式。默认情况下，它将把数据放在一个空目录中。本质上，它将连接到源服务器然后在网络连接上执行 tar，并且把数据放到想要的目录中。

这种方法的问题是 pg_basebackup 将创建很多文件，如果想要把这种备份移动到一种外部备份方案（可能是 Tivoli 存储管理器或者某种其他方案）就不合适：

```
-F, --format=p|t     output format (plain (default), tar)
```

要创建单个文件，可以使用“-F=t”选项。默认情况下，它将创建一个名为 base.tar 的文件，这样就更容易管理。当然，缺点是用户在执行 PITR 之前必须先展开该文件。

4. 测试事务日志

在我们深入到实际的重放处理之前，有必要检查归档以确保归档能正确地并且按照

预期工作：

```
[hs@zenbook archive]$ ls -l
total 212996
-rw------- 1 hs hs 16777216 Jan 30 09:04 000000010000000000000001
-rw------- 1 hs hs 16777216 Jan 30 09:04 000000010000000000000002
-rw------- 1 hs hs      302 Jan 30 09:04
000000010000000000000002.00000028.backup
-rw------- 1 hs hs 16777216 Jan 30 09:20 000000010000000000000003
-rw------- 1 hs hs 16777216 Jan 30 09:20 000000010000000000000004
-rw------- 1 hs hs 16777216 Jan 30 09:20 000000010000000000000005
-rw------- 1 hs hs 16777216 Jan 30 09:20 000000010000000000000006
...
```

只要在数据库上进行重要的活动，WAL 文件就应该被发送到归档。

除了检查文件之外，还有下面的视图可用：

```
test=# d pg_stat_archiver
             View "pg catalog.pg stat archiver"
       Column       |           Type           | Modifiers
--------------------+--------------------------+-----------
 archived_count     | bigint                   |
 last_archived_wal  | text                     |
 last_archived_time | timestamp with time zone |
 failed_count       | bigint                   |
 last_failed_wal    | text                     |
 last_failed_time   | timestamp with time zone |
 stats_reset        | timestamp with time zone |
```

当归档因为某种原因停转时，pg_stat_archiver 扩展可以用来找出原因。它将告诉用户已经被归档的文件数量（archived_count），用户也能看到哪个文件是最后一个文件以及它何时被归档。最后，pg_stat_archiver 扩展可以告诉用户归档什么时候发生错误，这是关键性信息。不幸的是，错误代码或者错误消息没有被显示在表中，但由于 archive_command 可以是任意命令，错误信息很容易记录下来。

在归档中其实还有更多东西，如上所述，有必要看看哪些文件已经实际被归档。但还不只如此，当 pg_basebackup 扩展被调用时，用户将在 WAL 文件流中看到一个.backup 文件。这个文件很小并且只包含有关基础备份本身的一些信息——它纯粹就是一种告知性文件，在重放处理中并不需要它。但是，这种文件给出了一些至关重要的线索。当用户后来开始重放事务日志时，用户可以删除所有比.backup 文件老的 WAL 文件。在这个案例中，我们的备份文件是 000000010000000000000002.00000028.backup。这表示重放处

理会从文件...0002 中的某处（位置...28）开始，它还表示我们可以删除所有比...0002 老的文件，恢复不再需要更旧的 WAL 文件。记住用户可以保留多于一个备份，因此笔者这里只会提到当前备份。

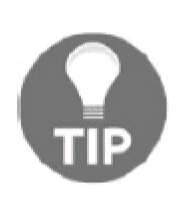
现在归档开始工作，我们可以把注意力转向重放处理。

10.2.4　重放事务日志

让我们总结一下到目前为止的整个过程。我们已经调整了 postgresql.conf 文件（wal_level、max_wal_senders、archive_mode 和 archive_command），并且我们已经在 pg_hba.conf 文件中允许了 pg_basebackup 扩展，然后数据库已经被重启过并且成功地产生了一个基础备份。

记住基础备份可以在数据库全面运转时发生——只需要一次快速的重启来改变 max_wal_sender 以及 wal_level 变量。备份可以在数据库正常提供服务时发生。

现在，在系统已经可以正确工作之后，可能会面临崩溃，并且希望从中恢复过来。因此，我们可以执行 PITR 以恢复尽可能多的数据。我们要做的第一件事情是拿到基础备份并且把它放在想要的位置。



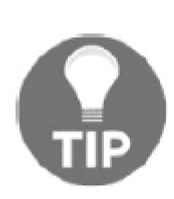
保存旧的数据库集簇是一种好办法。即使它损坏，你的 PostgreSQL 支持公司可能需要它来追踪崩溃的原因。在让一切恢复正常开始运行之后，还是可以删除它。

按照前面给定的文件系统布局，用户可能想要这样做：

```
cd /some target dir
cp -Rv * /data
```

笔者假定用户的新数据库服务器将被放在/data 目录中。在将基础备份复制过去之前，要确定该目录为空。

下一步将创建一个名为 recovery.conf 的文件。它将包含重放处理关心的所有信息，例如 WAL 归档的位置、想要达到的时间点等。



在 PostgreSQL 10.0 中，recovery.conf 很可能不再存在。那些设置预计被转移到 postgresql.conf 文件中。在写作本章时，笔者还没有完全确定到底会怎样处理。

下面是一个 recovery.conf 文件的例子：

```
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2019-04-05 15:43:12'
```

把 recovery.conf 文件放到$PGDATA 目录中后，用户可以简单地启动其服务器。输出可能像这样：

```
server starting
LOG: database system was interrupted; last known up
  at 2017-01-30 09:04:07 CET
LOG: starting point-in-time recovery to 2019-04-05 15:43:12+02
LOG: restored log file "000000010000000000000002" from archive
LOG: redo starts at 0/2000028
LOG: consistent recovery state reached at 0/20000F8
LOG: restored log file "000000010000000000000003" from archive
LOG: restored log file "000000010000000000000004" from archive
LOG: restored log file "000000010000000000000005" from archive
...
LOG: restored log file "00000001000000000000000E" from archive
cp:  cannot stat '/archive/00000001000000000000000F':
  No such file or directory
LOG: redo done at 0/E7BF710
LOG: last completed transaction was at log time
  2017-01-30 09:20:47.249497+01
LOG: restored log file "00000001000000000000000E" from archive
cp:  cannot stat '/archive/00000002.history': No such file or directory
LOG: selected new timeline ID: 2
cp:  cannot stat '/archive/00000001.history': No such file or directory
LOG: archive recovery complete
LOG: MultiXact member wraparound protections are now enabled
LOG: database system is ready to accept connections
LOG: autovacuum launcher started
```

在服务器被启动时，可以查找若干消息来确认恢复过程工作正确，第一个是 consistent recovery state reached。这个消息意味着 PostgreSQL 可以重放足够多的事务日志将数据库带回到一个状态，该状态会让数据库真正可用。

然后 PostgreSQL 将逐个复制文件并且重放它们。不过，记住我们已经告诉 recovery.conf 文件把我们一路带到 2019 年。但这本书写作于 2017 年，显然没有足够的 WAL 能到达 2019 年。因此，PostgreSQL 将会报错并且告诉我们有关最后一个已完成事务的情况。

当然这只是一个例子而已，现实世界中的例子中用户将很可能使用一个过去的日期，这样就能安全地用它来恢复。但是，笔者想要表达的是使用未来的日志是完全可行的——只是要做好准备面对将会发生的错误。

在恢复完成后，recovery.conf 文件将被重命名为 recovery.done，这样用户可以看到在恢复期间发生了什么。所有的数据库服务器进程都将被启动并且运行，并且用户就已经可以使用数据库实例了。

- 查找正确的时间戳

到目前为止，笔者都是假定用户已经知道要恢复到哪个时间戳或者用户只是想重放所有的事务日志以尽可能减少数据损失。但是，如果用户不想重放所有日志该怎么办，如果用户不知道要恢复到哪个时间点该怎么办？在日常生活中，这实际上是一种很常见的场景。一个开发人员在上午丢失了一些数据，现在想让一切都回到正轨。但问题是，上午的什么时候？一旦恢复结束，就不能很容易地重新开始一次。一旦恢复完成，系统就会被提升。而一旦系统被提升，用户就无法继续重放 WAL。

不过用户可以暂停恢复而不做提升，检查数据库中有什么，然后继续。

这样做比较容易。第一件要做的事情是保证 postgresql.conf 文件中 hot_standby 变量被设置为 on。这会让数据库在恢复模式中也可读，然后在开始重放处理之前还需要修改一下 recovery.conf 文件：

```
recovery_target_action = 'pause'
```

有好几种 recovery_target_action 设置。如果使用的是 pause，PostgreSQL 将在达到想要的时间点后暂停，让用户有机会检查哪些东西已经被重放。这时用户可以调整想要达到的时间，重新启动并且重试重放。此外，还可以把该值设置为 promote 或者 shutdown。

有第二种方法可以暂停事务日志重放。本质上，它也能在执行 PITR 时使用。但是，在大部分情况下它被用于流复制。在 WAL 重放期间可以使用下面的功能：

```
postgres=# x
Expanded display is on.
postgres=# df *pause*
List of functions
-[ RECORD 1 ]-------+-------------------------
Schema              | pg_catalog
Name                | pg_is_xlog_replay_paused
Result data type    | boolean
Argument data types |
Type                | normal
-[ RECORD 2 ]-------+-------------------------
Schema              | pg_catalog
```

```
Name                  | pg_xlog_replay_pause
Result data type      | void
Argument data types   |
Type                  | normal

postgres=# df *resume*
List of functions
-[ RECORD 1 ]-------+---------------------
Schema                | pg_catalog
Name                  | pg_xlog_replay_resume
Result data type      | void
Argument data types   |
Type                  | normal
```

用户可以调用“SELECT pg_xlog_replay_pause();”命令停住 WAL 重放，直到调用“SELECT pg_xlog_replay_resume();”命令再继续。

其想法是找出有多少 WAL 已经被重放并且根据需要继续重放。但是，要记住一点：一旦服务器被提升，如果不做一些预防工作，用户就无法继续重放 WAL。

如你所见，要找出已经恢复了多少可能会相当棘手。因此，PostgreSQL 提供了一些方法来协助。考虑下面的实际例子，一个午夜，用户正在运行每晚都会进行的处理并且结束于一个通常无法知晓的时间点，而目标是正好恢复到该处理的结束点。那么问题就来了：如何知晓该处理何时结束？在大部分情况下，这是很难找出的。所以不如在事务日志中加上一个标记：

```
postgres=# SELECT pg create restore point('my daily process ended');
 pg_create_restore_point
-------------------------
 1F/E574A7B8
(1 row)
```

如果用户的处理在一结束时就调用这个 SQL 语句，就可以使用事务日志中的这个标签来正好恢复到这个时间点，只需要把下列内容直接加到 recovery.conf 文件中：

```
recovery_target_name = 'my_daily_process_ended'
```

使用这个设置取代 recovery_target_time，重放处理把用户正好带回该操作的结束点。

当然，用户还可以重放到一个特定的事务 ID。不过，在实际生活中已经证明这很难做到，因为管理员更难准确地知道事务 ID，因此这种做法没有很大的实用价值。

10.2.5 清理事务日志归档

到目前为止，数据已经被不停地写入归档，但却没有关注对归档进行清理，以便在文件系统中释放一些空间。PostgreSQL 无法为用户做这件工作，因为它不知道用户是否还会再次使用这些归档。因此，用户需要负责清理事务日志。当然，用户也可以使用一种备份工具——但是，有必要知道 PostgreSQL 不会为用户执行清理。

假定我们想要清理掉旧的事务日志，它们不再被需要。用户可能想要保留几个基础备份，并且清理所有恢复那些备份时不再需要的事务日志。

在这种情况下，pg_archivecleanup 扩展正是我们所需要的。用户可以简单地把归档目录和备份文件的名称传递给 pg_archivecleanup 扩展，它确保把文件从磁盘上移除。使用这个工具可以省很多事，因为用户不必自行找出哪些事务日志文件需要被保留。该扩展的用法如下：

```
[hs@pgnode01 ~]$ pg_archivecleanup --help
pg_archivecleanup removes older WAL files from PostgreSQL archives.

Usage:
  pg_archivecleanup [OPTION]... ARCHIVELOCATION OLDESTKEPTWALFILE

Options:
  -d               generate debug output (verbose mode)
  -n               dry run, show the names of the files that
                       would be removed
  -V, --version    output version information, then exit
  -x EXT           clean up files if they have this extension
  -?, --help       show this help, then exit

For use as archive_cleanup_command in recovery.conf when standby_mode = on:
  archive_cleanup_command = 'pg_archivecleanup
         [OPTION]... ARCHIVELOCATION %r'
e.g.
  archive_cleanup_command = 'pg_archivecleanup
    /mnt/server/archiverdir %r'

Or for use as a standalone archive cleaner:
e.g.
  pg_archivecleanup /mnt/server/archiverdir
         000000010000000000000010.00000020.backup
```

该工具易于使用，并且在所有的平台上都可用。

10.3 设置异步复制

在看过了事务日志归档和 PITR 之后，我们可以把重心放在如今 PostgreSQL 世界中一种被广泛使用的特性上：流复制。流复制的思想非常简单：在一个初始的基础备份之后，次服务器可以连接到主服务器并且实时取得事务日志进行应用。事务日志重放不再是一个单一操作，而是一种连续处理，只要集群还存在这种处理就被假定会持续运行下去。

10.3.1 执行基本设置

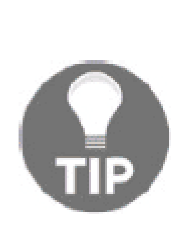
在本节中，读者将学到如何快速简单地设置异步复制。我们的目标是设置一个由两个节点组成的系统。

基本上，大部分的工作已经在设置 WAL 归档时完成了。但是为了读者容易理解，笔者还是会解释整个设置过程，因为我们不能假定 WAL 传送实际已经被按照需要设置好。

第一件事情是在 postgresql.conf 文件中调整下列参数：

```
wal_level = replica
max_wal_senders = 5     # or whatever value >= 2
hot_standby = on        # already a sophistication
```

和之前一样，wal_level 变量必须调整以确保 PostgreSQL 产生足够的事务日志来支撑一台从服务器，然后必须配置 max_wal_senders 变量。当一台从服务器启动运行或者一个基础备份被创建时，一个 WAL 发送进程将与客户端上的一个 WAL 接收进程进行对话。max_wal_senders 设置允许 PostgreSQL 创建足够多的发送进程来为那些客户端提供服务。



理论上，只有一个 WAL 发送进程就足够了。但那样会不太方便。一次使用--xlog-method=stream 参数的基础备份就已经需要两个 WAL 发送进程。如果用户想要运行一台从服务器并且同时执行一次基础备份，就已经需要使用 3 个进程。因此，要确保允许 PostgreSQL 创建足够多的进程，这样可以省下一些无谓的重启。

然后轮到 hot_standby 变量。基本上主服务器会忽略 hot_standby 变量并且不把它纳入考虑的范畴，它的全部作用是让从服务器在 WAL 重放期间可读。那么我们为何要关心它？记住这一点： pg_basebackup 扩展将会克隆整个服务器，包括其配置。这意味着如果用户已经在主服务器上设置好这个值，当数据目录被克隆时，从服务器将自动地得到它。

在设置好 postgresql.conf 文件后，我们可以转向 pg_hba.conf 文件：通过增加规则只

允许从服务器执行复制。那些规则基本上和在 PITR 时已经见过的规则一样。

然后像做 PITR 时那样重新启动数据库服务器。

接下来可以在从服务器上调用 pg_basebackup 扩展。在调用前，确保/target 目录为空。如果用户使用的是 RPM 包，请确认已经关闭可能运行着的实例并且清空该目录（例如/var/lib/pgsql/data）：

```
pg_basebackup -D /target
         -h master.example.com
         --checkpoint=fast
         --xlog-method=stream -R
```

只需要将/target 目录替换为想要的目标目录并且将 master.example.com 替换为主服务器的 IP 或者 DNS 名称即可。--checkpoint=fast 参数将会触发立刻执行一个检查点。然后是--xlog-method=stream 参数：它将开启两个流。一个将复制数据，另一个将在运行备份时被创建以取得 WAL。

最后是-R 标志：

```
-R, --write-recovery-conf
      write recovery.conf after backup
```

-R 标志实在是一种好特性，它让 pg_basebackup 扩展能自动创建从服务器的配置。它将对 recovery.conf 文件增加一些项：

```
standby_mode = on
primary_conninfo = ' ... '
```

第一个设置表示 PostgreSQL 应该一直重放 WAL——如果所有的事务日志都已经被重放，它应该等待新的 WAL 到来。第二个设置告诉 PostgreSQL 主服务器在哪里。它是一个普通的数据库连接。



从服务器也能连接到其他从服务器来流式接收事务日志。通过从一个从服务器创建基础备份可以级联复制。因此在这种上下文环境中，主服务器实际上意味着源服务器。

在运行了 pg_basebackup 扩展之后，服务就已经可以启动。第一件应该检查的事情是主服务器上是否出现了 WAL 发送进程：

```
[hs@linuxpc ~]$ ps ax | grep sender
17873 ?        Ss     0:00 postgres: wal sender process
                               ah ::1(57596) streaming 1F/E9000060
```

如果有一个 WAL 发送进程，那么从服务器也将运行着一个 WAL 接收进程：

```
17872 ?        Ss     0:00 postgres: wal receiver process
                                streaming 1F/E9000060
```

如果那些进程都已经出现，就说明我们已经走上了正轨并且复制已经按照预期开始工作。现在两端都已经开始进行对话，WAL 会从主服务器流向从服务器。

- 提高安全性

到目前为止，读者已经看到的是以超级用户流式传送数据。但是，允许来自远端的超级访问并不是什么好主意。幸运的是，PostgreSQL 允许创建一个只被允许消费事务日志流，而不能做任何其他事情的用户。

创建只用于流复制的用户很容易：

```
test=# CREATE USER repl LOGIN REPLICATION;
CREATE ROLE
```

通过将 replication 指派给该用户，就使得该用户只能被用于复制——做任何其他的事情都会被禁止。

笔者高度建议不要使用超级用户账户设置流复制，而是修改 recovery.conf 文件使用新建用户。避免暴露超级用户账户将极大地提高安全性（就像给复制用户一个口令）。

10.3.2　停止和继续复制

一旦流复制被设置好，它就会完美无瑕地工作而无须管理员过多地介入。但是，在某些情况下可能有必要停止复制并且在之后的某个时间继续它。为什么会有人想这样做？

考虑下面的用例，用户负责一个主/从设置，它运行着某种蹩脚的 CMS 或者某种靠不住的论坛软件。假定用户想要把应用从蹩脚的 CMS 1.0 更新为蹩脚的 CMS 2.0。在用户的数据库中将执行一些更改，它们将被立即复制到从数据库中。如果更新处理做错了什么会怎样？由于有流复制，错误将立即被复制到两个节点上。

为了避免立即复制，用户可以停止复制并且根据需要再继续。在 CMS 更新例子中，我们可能会做下面的事情：

- 停止复制。
- 在主服务器上执行应用更新。
- 检查应用是否仍在工作。如果是，继续复制。否则故障转移到复制品上，它仍具有旧数据。

使用这种机制用户可以保护其数据，因为用户可以退回到发生问题之前的数据。在

本章稍后的部分，读者将学到如何把一台从服务器提升为新的主服务器。

现在的主要问题是：如何能够停止复制？下面是方法，在后备机上执行下面的命令：

```
test=# SELECT pg_xlog_replay_pause();
```

这一命令将停止复制。注意事务日志将仍然从主服务器流向从服务器——只有重放处理被停止。用户的数据仍然受到保护，因为它已经被持久化在从服务器上。在服务器崩溃的情况下，不会有数据丢失。

记住在从服务器上的重放必须被停止，否则 PostgreSQL 将会抛出一个错误：

```
ERROR: recovery is not in progress
HINT: Recovery control functions can only be executed during recovery.
```

一旦流复制被继续，将需要在从服务器上执行下面的命令：

```
SELECT pg_xlog_replay_resume();
```

PostgreSQL 将再次开始重放 xlog。

10.3.3　检查复制以确保可用性

每一位管理员的核心任务之一是确保复制一直保持运转。如果复制停摆，主服务器崩溃时就有可能丢失数据。因此，留意复制是绝对有必要的。

幸运的是，PostgreSQL 提供了系统视图允许用户深入地查看系统的行为。其中之一是 pg_stat_replication：

```
test=# d pg_stat_replication
          View "pg_catalog.pg_stat_replication"
      Column       |           Type           | Modifiers
------------------+--------------------------+-----------
 pid              | integer                  |
 usesysid         | oid                      |
 usename          | name                     |
 application_name | text                     |
 client addr      | inet                     |
 client_hostname  | text                     |
 client_port      | integer                  |
 backend_start    | timestamp with time zone |
 backend_xmin     | xid                      |
 state            | text                     |
 sent_location    | pg_lsn                   |
 write_location   | pg_lsn                   |
```

```
flush_location   | pg_lsn                  |
replay_location  | pg_lsn                  |
sync_priority    | integer                 |
sync_state       | text                    |
```

pg_stat_replication 视图将包含发送者上的信息。笔者这里不想使用主服务器这个词，因为从服务器可能被连接在其他从服务器上，这样就有可能会构建一棵服务器树。在构成服务器树的情况下，主服务器将只有直接与它相连的从服务器的信息。

在这个视图中用户将会看到的第一件事情是 WAL 发送进程的进程 ID。在有事情出错时，它可以帮助用户识别进程（通常并非如此）。然后用户将看到从服务器用来连接到其发送服务器的用户名。client_域将指示从服务器在哪里，用户将能从这些域中提取到网络信息。之后是 backend_start 域，它显示从服务器何时开始从源服务器进行流传送。

之后是神奇的 backend_xmin 域。假定用户运行一个主/从设置，可以设置从服务器向主服务器报告从服务器的事务 ID。这样做的想法是延迟主服务器上的清理，使得数据不会被从运行在从服务器上的事务中夺走。

state 域告诉用户有关服务器的状态。如果系统没有问题，这个域将包含 streaming，否则需要更进一步的观察。

接下来的 4 个域很重要。sent_location 域表示有多少 WAL 已经到达另一端（已被 WAL 接收者接受）。用户可以使用它找出有多少数据已经到达了从服务器。然后是 write_location 域。一旦 WAL 已经被接受，它会被传递到 OS。write_location 域将告诉我们已经安全到达 OS 的 WAL 位置。flush_location 域将显示数据库已经把多少 WAL 刷到了磁盘上。

最后还有 replay_location 域。WAL 已经到达后备机的磁盘上这一事实并不表示 PostgreSQL 已经重放了该 WAL（对最终用户可见）。假定复制被暂停，数据仍将流向后备机，但它将在以后才被应用。replay_location 域将告诉用户有多少数据已经可见。

最后 PostgreSQL 告诉我们复制是同步的还是异步的。

现在问题是人们如何使用这个视图得到关键的信息？一种常见的用例是检查复制延迟。下面是做法：

```
SELECT  client_addr, pg_current_xlog_location() - sent_location AS diff
FROM    pg_stat_replication;
```

当在主服务器上运行这个命令时，pg_current_xlog_location 函数返回当前的事务日志位置。PostgreSQL 有一种用于事务日志位置的特殊数据类型，叫作 pg_lsn，甚至还有若干操作符可以用来从主服务器的 xlog 位置减掉从服务器的 xlog 位置。因此这里的这个视图描述了两台服务器之间以字节计的差值（复制延迟）。

虽然 pg_stat_replication 扩展只包含发送端的信息，但 pg_stat_wal_receiver 扩展可以

提供接收端上类似的信息：

```
test=# d pg_stat_wal_receiver
           View "pg_catalog.pg_stat_wal_receiver"
        Column         |           Type           | Modifiers
-----------------------+--------------------------+-----------
 pid                   | integer                  |
 status                | text                     |
 receive_start_lsn     | pg_lsn                   |
 receive_start_tli     | integer                  |
 received_lsn          | pg_lsn                   |
 received_tli          | integer                  |
 last_msg_send_time    | timestamp with time zone |
 last_msg_receipt_time | timestamp with time zone |
 latest_end_lsn        | pg_lsn                   |
 latest_end_time       | timestamp with time zone |
 slot_name             | text                     |
 conninfo              | text                     |
```

在 WAL 接收进程的 ID 之后，PostgreSQL 将为用户提供进程的状态。然后 receive_start_lsn 域将告知用户 WAL 接收进程从哪个事务日志位置开始，而 receive_start_tli 域将告诉我们 WAL 接收进程启动时使用的时间线。

received_lsn 域包含有关已经收到并且刷入磁盘的 WAL 位置的信息，然后我们可以看到一些有关时间的信息以及有关槽和连接的信息。

通常，很多人会觉得阅读 pg_stat_replication 扩展比 pg_stat_wal_receiver 扩展更容易，并且大部分工具都围绕 pg_stat_replication 扩展来构建。

10.3.4　执行故障转移以及理解时间线

一旦创建好主/从设置，通常它能完美无瑕地工作很长一段时间。但是任何事情都有可能失败，因此有必要了解如何用一个备份系统替换一个失效的服务器。

PostgreSQL 让故障转移和提升变得很容易。基本上，用户需要做的就是用 pg_ctl 参数告诉一个复制系统提升自己：

```
pg_ctl -D data_dir promote
```

该服务器将把自己从主服务器断开连接并且立即执行提升。记住，在被提升时，从服务器可能已经支持着数千的只读连接。PostgreSQL 一个很好的特性是在提升期间所有打开的连接将被转变成读写连接——甚至无须重新连接。

在提升一台服务器时，PostgreSQL 将增加时间线。如果用户设置的是一台全新的服务器，它会处于时间线 1 中；如果从该服务器克隆出一台从服务器，它将和主服务器处于相同的时间线。因此，两个系统都是时间线 1。如果该从服务器被提升为一台独立的主服务器，它将移动到时间线 2。

时间线对于 PITR 特别重要。假定用户已经在大约午夜时创建了一个基础备份。在 12.00 AM 时，从服务器被提升。在 03.00 PM 时某些东西崩溃，并且用户想要恢复到 02.00 PM。用户将启动以重放在基础备份之后被创建的事务日志，并且顺着用户想要的服务器的 xlog 流继续下去，因为两个节点在 12.00 AM 时发生了分叉。

时间线改变也可以在事务日志文件的名称中见到。这里是时间线 1 中一个 xlog 文件的例子：

```
0000000100000000000000F5
```

如果时间线切换到 2，新的文件名可能会是这样：

```
0000000200000000000000F5
```

如你所见，来自不同时间线的 xlog 文件理论上可能存在于同一个归档目录中。

10.3.5　管理冲突

现在读者已经学了很多有关复制的内容。接下来，有必要看看复制冲突。首先要了解的问题是冲突是如何发生的？

考虑下面的例子：

主 服 务 器	从 服 务 器
	BEGIN;
	SELECT ... FROM tab WHERE ...
	...运行...
DROP TABLE tab;	...冲突发生...
	...事务被允许继续 30 秒...
	...在超时前冲突被解决或者结束...

这里的问题是，主服务器不知道在从服务器上正运行着一个事务。因此，DROP TABLE 命令不会阻塞到读取事务消失为止。如果那两个事务在同一个节点上发生，阻塞当然会发生。但是，我们这里考虑的是两台服务器。DROP TABLE 命令将正常执行，并且一个从磁盘上删除那些数据的请求也通过事务日志到达从服务器。从服务器就遇到了麻烦：如果该表被从磁盘移除，SELECT 子句就必须死掉——如果从服务器在应用那些

xlog 前等待 SELECT 子句完成，它又可能完全落后于主服务器。

理想的解决方案是一种折中，它可以用一个配置变量控制：

```
max_standby_streaming_delay = 30s
          # max delay before canceling queries
                    # when reading streaming WAL;
```

其思想是在通过杀死从服务器上的查询解决冲突之前等待 30 秒。根据用户的应用，用户可能想要把这个变量改为更加激进或者更加缓和的设置。注意这 30 秒是留给整个复制流而不是单一的查询。可能会发生一个查询提早很多被杀死，因为一些其他的查询已经等待了一段时间。

虽然 DROP TABLE 命令是一种很明显的冲突，但是还有一些不那么明显的操作。下面是一个例子：

```
BEGIN;
...
DELETE FROM tab WHERE id < 10000;
COMMIT;
...
VACUUM tab;
```

让我们再次假设在从服务器上有一个长时间运行的 SELECT 子句。

这里的 DELETE 子句显然不是问题，因为它只是把行标记为删除——它还没有实际移除它们。提交也不是问题，因为它只是标记事务为完成。物理上，行还在那里。

当一个诸如清理的操作介入时问题就出现了，它将毁掉磁盘上的行。当然，那些更改将进入 xlog 并且最终达到从服务器，然后就又陷入了麻烦。

要防止标准 OLTP 负载导致的典型问题，PostgreSQL 开发组已经引入了一个配置变量：

```
hot_standby_feedback = off
          # send info from standby to prevent
                    # query conflicts
```

如果这个设置为打开，从服务器将定期发送最老的事务 ID 给主服务器，然后清理操作将知道在系统中某处运行着一个比较老的事务并且把清理延迟到可以安全清除那些行时再进行。事实上，当主服务器上有长事务时，hot_standby_feedback 参数也会导致同样的效果。

如你所见，hot_standby_feedback 参数默认为关闭。为什么会这样？这样做有很好的理由，如果它为关闭，从服务器不会对主服务器有实际的影响。事务日志的流式传送不会消耗很多 CPU 能力，这让流复制廉价且有效。不过，如果一台从服务器（甚至可能不

在用户的管控之下）保持事务打开太久，用户的主服务器可能要忍受延迟清理带来的表膨胀。在默认设置下，相比减少冲突，我们更不愿意看到表膨胀的情况。

让 hot_standby_feedback = on 通常将避免 99%的 OLTP 相关的冲突，如果用户的事务比若干毫秒还要长，这一点就尤其重要。

10.3.6　让复制更可靠

在本章中，读者已经看到设置复制很简单，不需要花费多大功夫。但是，总是会有一些能够带来挑战的极限情况。其中之一就是事务日志保留。

考虑下面的情景：

- 取得一个基础备份。
- 在备份后一个小时内没有事情发生。
- 从服务器启动。

记住主服务器并不太在乎从服务器的存在性。因此，从服务器启动所需要的事务日志可能已经不再存在于主服务器之上，因为它们可能已经被检查点所移除。问题是，要启动从服务器，就需要一次重新同步。在数 TB 数据库的情况下，这显然是个问题。

这个问题的一种可能的解决方案是使用 wal_keep_segments 设置：

```
wal_keep_segments = 0       # in logfile segments, 16MB each; 0 disables
```

默认情况下，PostgreSQL 会保留足够多的事务日志以应付意外崩溃，但保留的不是太多。通过 wal_keep_segments 设置，用户可以告诉服务器保留更多数据，这样即使一台从服务器落后了，它也能追赶上来。

有必要记住服务器不仅会因为本身过慢或者过忙而落后——很多情况下延迟的发生是由于网络太慢。假设用户正在一个 1TB 的表上创建索引，PostgreSQL 排序数据，并且在实际构建索引时把它也发送到事务日志。想象一下要在一条可能只能处理 1G 比特的线路上发送数百兆字节的 xlog 意味着什么，数秒之内还会有很多数 GB 的数据可能随后到来。因此，调整 wal_keep_segments 设置不应局限于典型的延迟而是管理员可容忍的最高延迟（可能是某种安全范围）。

为 wal_keep_segments 研究一种较高且合理的设置很有意义，笔者推荐确保总是有足够的数据。

事务日志耗尽问题的另一种解决方案是复制槽，将在本章后面的部分介绍。

10.4　升级到同步复制

到目前为止，我们已经比较详细地介绍了异步复制。不过，异步复制意味着允许从

服务器上的提交在主服务器上的提交之后发生。如果主服务器崩溃，即便使用复制，还没有到达从服务器的数据仍有可能丢失。

同步复制可以解决这个问题，如果 PostgreSQL 同步进行复制，一次提交只有被至少一个复制系统刷入磁盘后才能在主服务器上完成。因此，同步复制本质上能大幅度降低数据丢失的几率。

在 PostgreSQL 中，配置同步复制也很容易。基本上，只需要做两件事情：

- 调整主服务器上 postgresql.conf 文件中的 synchronous_standby_names 设置。
- 为复制系统的 recovery.conf 文件中的 primary_conninfo 参数增加一个 application_name 设置。

让我们从主服务器上的 postgresql.conf 文件开始：

```
synchronous_standby_names = ''
         # standby servers that provide sync rep
                  # number of sync standbys and comma-separated
         # list of application_name
                  # from standby(s); '*' = all
```

如果放入的是“*”，所有节点都将被作为同步候选。不过，在实际生活场景中很可能只有若干个节点将被列出。例如：

```
synchronous_standby_names = 'slave1, slave2, slave3'
```

现在我们必须更改 recovery.conf 文件并且增加 application_name：

```
primary_conninfo = '... application_name=slave2'
```

这个复制系统将作为 slave2 连接到主服务器。主服务器将检查其配置并且发现 slave2 是列表中列出的可行从服务器之一。因此，PostgreSQL 就能保证只有在从服务器确认拿到一个事务后该事务才能在主服务器上成功提交。

现在假设 slave2 由于某种原因宕机，PostgreSQL 将尝试把其他两个节点之一转变成一个同步后备机。现在的问题是，如果没有其他的服务器会怎样？在这种情况下，如果事务被认为是同步的，PostgreSQL 将一直等待提交。是的，就是这样，PostgreSQL 将不会继续提交，除非有至少两个可行节点可用。记住，用户已经要求 PostgreSQL 在至少两个节点上存储数据——如果在任意给定时间点用户无法提供足够多的主机，那是用户的错。实际上，这意味着同步复制的最佳实现是至少 3 个节点，因为这样总是可以有损失一台主机的机会。

谈到主机失效，有必要提到一点，如果一个同步伙伴在提交进行时死掉，PostgreSQL 将等待它回来。此外，同步提交可能会发生在某个其他潜在的同步伙伴上。最终用户甚

至可能不会意识到同步伙伴发生了变化。

在一些情况下，仅在两个结点上保存数据可能不够，也许用户想要更多地提高安全性并且在更多节点上存储数据。为了实现这一点，用户可以使用下面的语法（在 PostgreSQL 9.6 或更高版本中）：

```
synchronous standby names =
   '4(slave1, slave2, slave3, slave4, slave5, slave6)'
```

在这种情况下，在主服务器确认提交之前，数据被认为会落在 6 个节点中的 4 个之上。

当然，这样做是有代价的——记住如果用户增加越来越多的同步复制系统，速度将会下降，天下没有免费的午餐。为了保持性能开销可控，PostgreSQL 提供了若干方法，在接下来的小节中将会讨论它们。

● 调整持久性

在本章中，读者已经看到数据可以被同步地或者异步地复制。不过，这并非流复制的全部。为了确保好的性能，PostgreSQL 允许用户以非常灵活的方式配置复制中涉及的各类事情。可以同步或者异步复制所有东西，但是在很多情况下用户可能想要以更细粒度的方式来运行，这时就需要 synchronous_commit 设置。

假定已经配置了同步复制（ recovery.conf 文件中的 application_name 设置以及 postgresql.conf 文件中的 synchronous_standby_names 设置），synchronous_commit 设置将提供下列选项。

- off：这本质上就是异步复制。在主服务器上 WAL 不会被立即刷到磁盘，并且主服务器不会等待从服务器把所有东西都写到磁盘。如果主服务器失效，一些数据可能会丢失（最多是 wal_writer_delay 的 3 倍）。
- local：主服务器上提交时事务日志被刷到磁盘。但是，主服务器不等待从服务器（异步复制）。
- remote_write：remote_write 设置就已经能让 PostgreSQL 进行同步复制。不过，只有主服务器会把数据保存到磁盘。对于从服务器，把数据发送到操作系统就够了。其思想是不要等待第二次盘刷写以提高速度。两种存储系统正好在同一时间崩溃的可能性非常低。因此，数据损失的风险近乎于零。
- on：在这种情况下，如果主服务器和从服务器都已经成功地把一个事务刷到了磁盘，那么这个事务就没有问题了。只有数据被安全地保存在两台服务器上（或者更多服务器，取决于配置）后，应用才会收到提交成功的通知。
- remote_apply：虽然 on 能确保数据被安全地存储在两个节点上，但它不保证能够马上做到负载均衡。数据被刷到磁盘上并不能确保用户已经可以看到该数

据。例如，如果存在冲突，从服务器将停止事务重放——然后，冲突期间事务日志仍将被发送给从服务器并且被刷到磁盘上。简而言之，可能会发生数据被刷到从服务器上但对最终用户还不可见的情况。remote_apply 能解决这一问题。它确保数据必须在复制系统上可见，这样接下来的读请求可以被安全地在从服务器上执行，这些请求已经能够看到对主服务器所作的更改并且把更改显示给最终用户。

当然，remote_apply 是一种最慢的复制数据的方法，因为它要求等待从服务器把数据披露给最终用户。

在 PostgreSQL 中，synchronous_commit 参数不是一个全局值。就像很多其他设置一样，可以在多个级别上调整这个参数。用户可能会这样做：

```
test=# ALTER DATABASE test SET synchronous_commit TO off;
ALTER DATABASE
```

有时候，只有单个数据库应该以特定的方式进行复制。还可以在作为特定用户连接时只做同步复制。最后但并非最不重要的，还可以告诉单个事务如何提交。通过即时调整 synchronous_commit 参数，甚至可以以每个事务的级别进行控制。

例如，考虑下面的两种场景：

- 写入一个日志表，在其中用户可能想要使用异步提交，因为希望更快。
- 存储一次信用卡付款，在其中可能希望更安全，因此用户可能想要使用同步事务。

如你所见，根据什么数据被修改，同一个数据库可能有不同的需求。因此，在事务级别更改数据非常有用并且有助于提高速度。

10.5　利用复制槽

在介绍了同步复制和动态可调整的持久性之后，笔者想要把读者的注意力转到一种被称为复制槽的特性上。

复制槽的目的是什么？让我们考虑下面的例子：有一台主服务器和一台从服务器。在主服务器上，执行一个大型的事务并且网络连接不足以及时传送所有的数据。在某时刻，主服务器移除了它的事务日志（检查点）。如果从服务器落后太多，就需要一次重新同步。正如在前文已经见到的，wal_keep_segments 设置可以被用来降低复制失败的风险。但问题是，wal_keep_segments 设置的最佳值是多少？当然，越多越好，但多少是最好？

复制槽将为用户解决这一问题：如果用户使用复制槽，主服务器只能在事务日志被所有复制系统消费过后重用它。其优点是从服务器绝不会落后太多以至于需要重新同

步。但问题是，假定用户在没有告知主服务器的情况下关闭了一个复制系统。主服务器将永远保持事务日志并且主服务器上的磁盘将最终填满导致不必要的停机。

为了降低主服务器的风险，复制槽应该只和适当的监控以及告警措施一起使用。有必要关注打开的复制槽，它们可能会导致问题或者可能没有继续被使用。

在 PostgreSQL 中有两种类型的复制槽：

- 物理复制槽。
- 逻辑复制槽。

物理复制槽可以被用于标准的流复制。它们将确保数据不会被过早重用。逻辑复制槽做的是相同的事情，但它们被用于逻辑解码。逻辑解码的思想是让用户有机会挂接到事务日志并且用一个插件对它解码。因此逻辑事务槽可以看作是对数据库实例做了某种 tail -f。它允许提取对数据库做的更改，并且可以输出成任意格式的事务日志并且可用于任何目的。在很多情况下逻辑复制槽被用于逻辑复制。

10.5.1　处理物理复制槽

要使用复制槽，必须对 postgresql.conf 文件做出更改：

```
wal_level = logical
max_replication_slots = 5      # or whatever number is needed
```

有了物理槽，逻辑就不是必需的了——replica 就足够了。不过，对于逻辑槽，我们需要更高的 wal_level 设置，然后必须更改 max_replication_slots 设置。说白了，只需要放入一个服务用户目的所需的数字就好。笔者的推荐是加上一些空闲的槽，这样可以轻易地接入更多消费者而无须重启服务器。

在重启后，槽就已经可以被创建了：

```
test=# x
Expanded display is on.
test=# df *create*physical*slot*
List of functions
-[ RECORD 1 ]-------+-----------------------------------------------
----------------
Schema              | pg_catalog
Name                | pg_create_physical_replication_slot
Result data type    | record
Argument data types | slot_name name,
                      immediately reserve boolean DEFAULT false,
                      OUT slot_name name,
```

```
                              OUT xlog_position pg_lsn
Type                         | normal
```

这里的 pg_create_physical_replication_slot 函数帮助用户创建槽。它可以用一个或者两个参数调用：在只传入一个槽名称的情况下，槽将在第一次被使用时激活。如果 true 被作为第二个参数传入，槽将立即开始保留事务日志：

```
test=# SELECT *
         FROM pg_create_physical_replication_slot('some_slot_name', true);
   slot name     | xlog position
----------------+---------------
 some_slot_name | 0/EF8AD1D8
(1 row)
```

要查看主服务器上有多少个活动的槽，考虑运行下面的 SQL 语句：

```
test=# x
Expanded display is on.
test=# SELECT * FROM pg replication slots;
-[ RECORD 1 ]-------+---------------
slot_name           | some_slot_name
plugin              |
slot_type           | physical
datoid              |
database            |
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         | 0/EF8AD1D8
confirmed_flush_lsn |
```

该视图将告诉我们很多有关槽的信息。它包含有关使用中的槽的类型、事务日志位置等的信息。

要使用槽，所需要做的就是把它加入 recovery.conf 文件中：

```
primary_slot_name = 'some_slot_name'
```

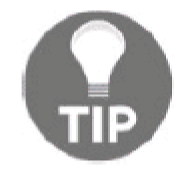
一旦流被重启，槽将被直接使用并且保护复制。

如果用户不再需要槽，可以很容易地删掉它：

```
test=# df *drop*slot*
List of functions
```

```
-[ RECORD 1 ]-------+------------------------
Schema              | pg_catalog
Name                | pg_drop_replication_slot
Result data type    | void
Argument data types | name
Type                | normal
```

在槽被删除时，就不再有逻辑槽和物理槽的区分。只需向该函数传递槽的名称并且执行它即可。



注意当槽被删除时，不允许任何人使用它。否则，PostgreSQL 将会报错（有充分的理由）。

10.5.2　处理逻辑复制槽

逻辑复制槽对于逻辑复制来说极其重要。不幸的是，由于篇幅限制，本章无法覆盖到逻辑复制的所有方面。不过，笔者想要勾勒出一些基本概念，它们对于逻辑解码和逻辑复制都非常重要。

如果用户想要创建一个复制槽，其做法如下面的示例。这里需要的函数需要两个参数，第一个参数定义复制槽的名称，而第二个参数指定用于解码事务日志的插件。它决定 PostgreSQL 用来返回数据的格式：

```
test=# SELECT *
   FROM pg_create_logical_replication_slot('logical_slot',
'test_decoding');
  slot_name    | xlog_position
--------------+---------------
 logical_slot | 0/EF8AD4B0
(1 row)
```

用户可以使用与前文相同的命令检查现有的槽。

为了向读者展示槽实际能做什么，我们先创建一个小的测试：

```
test=# CREATE TABLE t_demo (id int, name text, payload text);
CREATE TABLE
test=# BEGIN;
BEGIN
test=# INSERT INTO t_demo VALUES (1, 'hans', 'some data');
INSERT 0 1
test=# INSERT INTO t_demo VALUES (2, 'paul', 'some more data');
```

```
INSERT 0 1
test=# COMMIT;
COMMIT
test=# INSERT INTO t_demo VALUES (3, 'joe', 'less data');
INSERT 0 1
```

注意有两个事务会被执行。这些事务所作的更改现在可以从槽中提取出来：

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
                                                pg_logical_slot_get_changes
------------------------------------------------------------------------
-----------------------------------
 (0/EF8AF5B0,606546,"BEGIN 606546")
 (0/EF8CCCA0,606546,"COMMIT 606546")
 (0/EF8CCCD8,606547,"BEGIN 606547")
 (0/EF8CCCD8,606547,"table public.t demo: INSERT: id[integer]:1
    name[text]:'hans' payload[text]:'some data'")
 (0/EF8CCD60,606547,"table public.t demo: INSERT: id[integer]:2
    name[text]:'paul' payload[text]:'some more data'")
 (0/EF8CCDE0,606547,"COMMIT 606547")
 (0/EF8CCE18,606548,"BEGIN 606548")
 (0/EF8CCE18,606548,"table public.t demo: INSERT: id[integer]:3
    name[text]:'joe' payload[text]:'less data'")
 (0/EF8CCE98,606548,"COMMIT 606548")
(9 rows)
```

这里使用的格式取决于之前我们选择的输出插件。PostgreSQL 有多种输出插件，例如 wal2json 之类。

注意在使用默认值的情况下，逻辑流将包含真实值而不只是函数。逻辑流中有最终出现在底层表中的数据。

还要记住槽中的数据一旦被消费掉就再也不会被返回：

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
 pg logical slot get changes
-----------------------------
(0 rows)
```

因此，第二次调用的结果集为空。如果用户想重复取得数据，PostgreSQL 提供了 pg_logical_slot_peek_changes 函数。其作用类似于 pg_logical_slot_get_changes 函数，但它能保证数据在槽中仍然可用。

当然，使用纯 SQL 并非消费事务日志的唯一方式。还有一个名为 pg_recvlogical 扩展

的工具。它的功能可以类比为在一整个数据库实例上做 tail -f 并且实时接收数据流。

让我们启动 pg_recvlogical 扩展：

```
[hs@zenbook ~]$ pg_recvlogical -S logical_slot -P test_decoding
   -d test -U postgres --start -f -
```

在这种情况下，该工具连接到测试数据库并且消费来自于 logical_slot 的数据。-f - 意味着流将被发送到 stdout。

让我们删掉一些数据：

```
test=# DELETE FROM t demo WHERE id < random()*10;
DELETE 3
```

这些更改将会进入事务日志。但是，数据库默认只关心删除后表会是什么样。它知道必须触碰哪些块等——它不知道以前是什么：

```
BEGIN 606549
table public.t_demo: DELETE: (no-tuple-data)
table public.t demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
COMMIT 606549
```

因此，这种数据几乎没有意义。为了弥补这一点，可以使用下面的命令：

```
test=# ALTER TABLE t_demo REPLICA IDENTITY FULL;
ALTER TABLE
```

如果该表被重新填入数据并且被再次删除，事务日志流会像这样：

```
BEGIN 606558
table public.t_demo: DELETE: id[integer]:1 name[text]:'hans'
    payload[text]:'some data'
table public.t_demo: DELETE: id[integer]:2 name[text]:'paul'
    payload[text]:'some more data'
table public.t demo: DELETE: id[integer]:3 name[text]:'joe'
    payload[text]:'less data'
COMMIT 606558
```

现在可以看到所有的更改。

- 逻辑槽用例

有多种复制槽的用例。最简单的用例如下，数据可以被从服务器以想要的格式得到并且被用于审计、调试或者监控数据库实例。

下一个逻辑步骤当然是取得更改流并且把它用于复制。BDR 之类的解决方案完全依赖于逻辑解码，因为二进制级别的更改在多主服务器复制的情况下无法工作。

最后，有时会需要在不停机的情况下进行升级。记住，二进制事务日志流不能被用来在不同版本的 PostgreSQL 之间进行复制。因此，未来版本的 PostgreSQL 将支持一种名为 pglogical 的工具，它可以帮助进行不停机升级。

10.6　总　　结

在本章中，读者已经学到了 PostgreSQL 复制的大部分重要特性，例如流复制和复制冲突。读者已经学到了 PITR 和复制槽。注意除非一本关于复制的书超过 400 页，否则它绝不算完整。但是读者已经学到了每一个管理员应该知道的最重要的事情。

第 11 章将介绍对 PostgreSQL 有用的扩展。读者将学到已经被工业界广泛采用的扩展，以及提供更多功能的扩展。

第 11 章　选定有用的扩展

在本书的第 10 章中，我们关注了复制、事务日志传输以及逻辑解码。这些内容大多与管理相关，现在我们的目标是介绍更加广泛的主题。在 PostgreSQL 世界中，很多事情都通过扩展完成。扩展的优点是可以在不膨胀 PostgreSQL 核心的前提下为核心增加功能。人们可以从功能相似的扩展中进行选择并且找到最适合自己的那一个。

本章将讨论一些 PostgreSQL 最广泛使用的扩展。不过，在深入这些内容前笔者做一点声明：本章只是介绍了笔者个人觉得有用的扩展，现在还有很多模块，本书不可能以一种合适的方式将它们全部覆盖。每天都有东西被发布出来，有时甚至连专业人员都很难注意到全部的扩展。

本章将覆盖下列主题：

- 扩展如何工作。
- 一批 contrib 模块。
- GIS 相关模块速览。
- 其他有用的扩展。

注意本章只会覆盖最重要的扩展。

11.1　理解扩展如何工作

在深入可用的扩展之前，有必要首先看看扩展如何工作，理解扩展机制的内部工作方式是非常有益处的。

首先看看语法：

```
test=# \h CREATE EXTENSION
Command:     CREATE EXTENSION
Description: install an extension
Syntax:
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ FROM old_version ]
             [ CASCADE ]
```

当用户想要部署一个扩展时，只要简单地调用 CREATE EXTENSION 子句。它将检查该扩展并且把扩展载入用户的数据库中。注意，扩展将被载入一个数据库而不是整个数据库实例。

如果用户装载一个扩展，用户可以决定要使用哪一个方案。很多扩展都能被重定位，因此用户可以选择使用的方案。然后可以选定扩展的一个特定版本，常常会有用户不想部署最新版本扩展的情况，因为其客户端还运行着过时的软件。在这种情况下，在系统上部署任意可用版本的能力可能会带来便利。

FROM old_version 子句需要多加注意。在旧时代，PostgreSQL 并不支持扩展，因此仍有很多未打包的代码存在。这个选项让 CREATE EXTENSION 子句能够运行一个安装脚本来把现有的对象吸收到扩展中，而不是创建新的对象。注意，SCHEMA 子句指定包含这些预先存在对象的方案，只有在拥有旧模块时才需要使用它。

最后还有 CASCADE 子句。一些扩展依赖于其他扩展，CASCADE 选项也会自动地部署那些软件包。例如：

```
test=# CREATE EXTENSION earthdistance;
ERROR: required extension "cube" is not installed
HINT: Use CREATE EXTENSION ... CASCADE to install required extensions too.
```

earthdistance 是一个实现了大圆距离计算的模块。读者可能知道，地面上两点间的最短距离并非直线。恰恰相反，飞行员必须不断地调整其航向以寻找从一点飞往另一点的最快路径。问题是：earthdistance 扩展依赖于 cube 扩展，后者允许用户在球体上执行操作。

为了自动部署这种依赖，如前所述，可以使用 CASCADE 子句：

```
test=# CREATE EXTENSION earthdistance CASCADE;
NOTICE: installing required extension "cube"
CREATE EXTENSION
```

在这种情况下，两个扩展都将被部署。

● 检查可用的扩展

PostgreSQL 提供了多个视图来确定系统中有哪些扩展以及哪些实际被部署。其中之一是 pg_available_extensions：

```
test=# \d pg_available_extensions
View "pg_catalog.pg_available_extensions"
      Column        | Type | Modifiers
--------------------+------+-----------
 name               | name |
```

```
 default_version   | text |
 installed_version | text |
 comment           | text |
```

它包含了所有可用扩展的列表，包括它们的名称、默认版本以及当前安装的版本。为了让最终用户更容易使用它，其中还有一段描述告诉我们更多有关该扩展的信息。

下面的列表包含两个从 pg_available_extensions 中取出的行：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_available_extensions LIMIT 2;
-[ RECORD 1 ]-----+-----------------------------------------------
-----
name              | earthdistance
default_version   | 1.1
installed_version | 1.1
comment           | calculate great-circle distances on the surface of
                    the Earth
-[ RECORD 2 ]-----+-----------------------------------------------
-----
name              | plpgsql
default_version   | 1.0
installed_version | 1.0
comment           | PL/pgSQL procedural language
```

如你所见，在笔者的数据库中 earthdistance 和 plpgsql 扩展都被启用。plpgsql 扩展默认就在那里而 earthdistance 则是刚刚和 cube 一起被加入的。这个视图的一个好处是用户可以很快地得到安装了什么以及可以安装什么的总览。

但是在一些情况下，扩展的可用版本不止一个。为了找出更多有关版本的信息，可以考虑检查下面的视图：

```
test=# \d pg_available_extension_versions
View "pg_catalog.pg_available_extension_versions"
   Column    |   Type   | Modifiers
-------------+---------+-----------
 name        | name    |
 version     | text    |
 installed   | boolean |
 superuser   | boolean |
 relocatable | boolean |
```

```
 schema        | name     |
 requires      | name[]   |
 comment       | text     |
```

如下一个列表所示，其中还有更多详细的信息可用：

```
test=# SELECT * FROM pg available extension versions LIMIT 1;
-[ RECORD 1 ]-------------------------------------------------------
name          | earthdistance
version       | 1.1
installed     | t
superuser     | t
relocatable | t
schema        |
requires      | {cube}
comment       | calculate great-circle distances on the surface of the
                Earth
```

PostgreSQL 还告诉用户扩展是否可以被重定位，它被部署在哪个方案中以及需要其他哪些扩展。然后有描述该扩展的注释，这个之前已经展示过。

现在的主要问题是，PostgreSQL 从哪里找到有关系统上扩展的所有信息？假定用户已经从官方的 PostgreSQL RPM 仓库部署了 PostgreSQL 9.6，/usr/pgsql-9.6/share/extension 目录将包含若干文件：

```
...
-bash-4.3$ ls -l citext*
-rw-r--r-- 1 root root   1028 Oct 26 13:28 citext--1.0--1.1.sql
-rw-r--r-- 1 root root   2748 Oct 26 13:28 citext--1.1--1.2.sql
-rw-r--r-- 1 root root    307 Oct 26 13:28 citext--1.2--1.3.sql
-rw-r--r-- 1 root root  12991 Oct 26 13:28 citext--1.3.sql
-rw-r--r-- 1 root root    158 Oct 26 13:28 citext.control
-rw-r--r-- 1 root root   9781 Oct 26 13:28 citext-unpackaged--1.0.sql
...
```

citext（大小写不敏感文本）扩展的默认版本为 1.3，因此有一个文件名为 citext--1.3.sql。此外，还有用来从一个版本转移到下一个版本（1.0--1.1、1.1--1.2 等）的文件。

然后还有一个.control 文件：

```
-bash-4.3$ cat citext.control
# citext extension
comment = 'data type for case-insensitive character strings'
```

```
default_version = '1.3'
module_pathname = '$libdir/citext'
relocatable = true
```

它包含与这个扩展相关的所有元数据，第一项包含注释，注意这一内容不会被显示在刚刚讨论过的系统视图中。当用户访问这些视图时，PostgreSQL 将进入这个目录并且读取所有的.control 文件，然后是默认版本和二进制文件的路径。如果用户从 RPM 安装一个典型的扩展，该目录将会是$libdir，它位于用户的 PostgreSQL 二进制目录中。不过，如果用户编写了自己的商业扩展，它很可能位于某个地方。

最后一个设置将告诉 PostgreSQL 该扩展是否可以放在任意一个方案中或者它是否必须放在一个固定的、预定义的方案中。

最后，还有散装文件。下面是其内容的一个摘录：

```
...
ALTER EXTENSION citext ADD type citext;
ALTER EXTENSION citext ADD function citextin(cstring);
ALTER EXTENSION citext ADD function citextout(citext);
ALTER EXTENSION citext ADD function citextrecv(internal);
...
```

散装文件将把现有的代码转变成一个扩展。因此，有必要在数据库中整合好现有的东西。

11.2　利用 contrib 模块

在对扩展进行了理论介绍之后，接下来看看一些最重要的扩展。在本节中，读者将了解到作为 PostgreSQL 的 contrib 模块提供给用户的模块。当用户安装 PostgreSQL 时，笔者推荐用户总是安装那些 contrib 模块，因为它们包含可以让用户工作更加便利的重要扩展。

在本节中，读者将见识到其中一些笔者认为最有趣的模块。

11.2.1　使用 adminpack

adminpack 模块的想法是让管理员可以不通过 SSH 来访问文件系统。这个包含有若干函数让这种访问成为可能。

为了把该模块载入数据库，可运行下面的命令：

```
test=# CREATE EXTENSION adminpack;
CREATE EXTENSION
```

adminpack 模块最有趣的特性之一是检查日志文件的能力。pg_logdir_ls 函数检查日志目录并且返回日志文件的列表：

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
ERROR: the log_filename parameter must
     equal 'postgresql-%Y-%m-%d_%H%M%S.log'
```

这里的重点是 log_filename 参数必须被调整为 adminspack 模块所需要的值。如果用户正好运行的是从 PostgreSQL 仓库下载的 RPM，log_filename 参数被定义为 postgresql-%a，在这种情况下它就需要更改以避免错误。

在更改后，一个日志文件名的列表会被返回：

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
          a          |                   b
---------------------+--------------------------------------------
 2017-03-03 16:32:58 | pg_log/postgresql-2017-03-03_163258.log
(1 row)
```

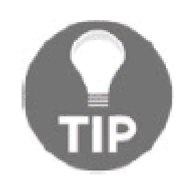
也可以确定磁盘上一个文件的尺寸。例如：

```
test=# SELECT b, pg_catalog.pg_file_length(b)
     FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
                    b                    | pg_file_length
-----------------------------------------+----------------
 pg_log/postgresql-2017-03-03_163258.log |           1525
(1 row)
```

除那些特性之外，该模块还提供了更多函数：

```
test=# SELECT proname FROM pg_proc WHERE proname ~ 'pg_file_.*';
    proname
----------------
 pg_file_write
 pg_file_rename
 pg_file_unlink
 pg_file_read
 pg_file_length
(5 rows)
```

用户可以读取、写入、重命名或者删除文件。



那些函数当然只能由超级用户调用。

11.2.2　应用布隆过滤器

PostgreSQL 9.6 开始可以使用扩展，随时增加索引类型。新的 CREATE ACCESS METHOD 命令以及一些额外的特性使得随时增加全功能且被事务日志记录的索引类型成为可能。

bloom 扩展为 PostgreSQL 用户提供了布隆过滤器，它是一种预滤器，能够有助于尽快地减小数据量。布隆过滤器的思想是计算一个位掩码并且把位掩码与查询对比。布隆过滤器可能会产生一些伪肯定但是仍然能极大地降低数据量。

当表由数百列和数百万行构成时，布隆过滤器就特别有用。由于不可能用 B-树来索引数百列，布隆过滤器是一种很好的替代品，因为它允许一次索引所有的东西。

为了便于展示，笔者安装了该扩展：

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

下一步，创建一个包含很多列的表：

```
test=# CREATE TABLE t bloom
(
   id           serial,
   col1         int4 DEFAULT random() * 1000,
   col2         int4 DEFAULT random() * 1000,
   col3         int4 DEFAULT random() * 1000,
   col4         int4 DEFAULT random() * 1000,
   col5         int4 DEFAULT random() * 1000,
   col6         int4 DEFAULT random() * 1000,
   col7         int4 DEFAULT random() * 1000,
   col8         int4 DEFAULT random() * 1000,
   col9         int4 DEFAULT random() * 1000
);
CREATE TABLE
```

为了让例子更加容易操作，那些列都有一个默认值，这样可以使用一个简单的 SELECT 子句增加数据：

```
test=# INSERT INTO t_bloom (id) SELECT *
     FROM    generate_series(1, 1000000);
INSERT 0 1000000
```

这个查询把一百万行加入该表中。现在该表可以被索引：

```
test=# CREATE INDEX idx_bloom ON t_bloom
     USING bloom(col1, col2, col3, col4, col5, col6, col7, col8, col9);
CREATE INDEX
```

注意该索引一次包含了 9 列。与 B-树不同，那些列的顺序实际不会造成区别。注意笔者刚创建的表在没有索引的情况下大约为 65MB。

索引在存储占用中又增加了另外的 15MB。

```
test=# \di+ idx_bloom
                        List of relations
 Schema |    Name    | Type  | Owner |  Table   | Size  | Description
--------+-----------+-------+-------+----------+-------+-------------
 public | idx_bloom | index | hs    | t_bloom | 15 MB |
(1 row)
```

布隆过滤器的好处是可以查找列的任意组合：

```
test=# explain SELECT count(*)
     FROM t_bloom
     WHERE col4 = 454
           AND col3 = 354
           AND col9 = 423;
                                  QUERY PLAN
-----------------------------------------------------------------------
-----------
 Aggregate (cost=20352.02..20352.03 rows=1 width=8)
   -> Bitmap Heap Scan on t_bloom (cost=20348.00..20352.02
                                             rows=1 width=0)
        Recheck Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
        -> Bitmap Index Scan on idx_bloom
             (cost=0.00..20348.00 rows=1 width=0)
             Index Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
(5 rows)
```

目前我们所看到的效果感觉上有点非同一般，那么很自然就会出现一个问题：为什么不总是使用布隆过滤器？原因很简单——为了使用布隆过滤器，数据库必须读取整个索引。而在 B-树的情况中则不必如此。

未来，很可能将会加入更多的索引类型以确保 PostgreSQL 能够覆盖更多的用例。

如果读者想要阅读更多有关布隆过滤器的内容，可以考虑阅读我们的博文：http://www.cybertec.at/trying-out-postgres-bloom-indexes/。

11.2.3　部署 btree_gist 和 btree_gin

在这一小节有关索引的简单介绍之后，会有更多索引相关的特性能被加入系统中。在 PostgreSQL 中有操作符类的概念，它们已经在第 5 章中讨论过。contrib 模块提供了两个扩展（即 btree_gist 和 btree_gin）来为 GiST 和 GIN 索引增加 B-树功能。

为什么它们这么有用？GiST 索引提供了多种 B-树不支持的特性，其中之一是执行 k 近邻（KNN）搜索的能力。

k 近邻搜索有什么用呢？ 想象某人在查找昨天中午左右增加的数据，那么到底是什么时候？在某些情况下可能很难给出界限。或者某人正在查找一种价格大约 70 欧元的产品。KNN 就可以解决这类查询。下面是一个例子：

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
```

在接下来的步骤中，一些简单的数据被加入：

```
test=# INSERT INTO t test SELECT * FROM generate series(1, 100000);
INSERT 0 100000
```

现在可以增加扩展：

```
test=# CREATE EXTENSION btree gist;
CREATE EXTENSION
```

为列增加一个 GiST 索引很容易。只要使用 USING gist 子句就好。注意在一个整数列上增加 GiST 索引只有在已经加入了该扩展时才能用；否则，PostgreSQL 将报告没有合适的操作符类：

```
test=# CREATE INDEX idx id ON t test USING gist(id);
CREATE INDEX
```

一旦索引被部署，就可以用距离来排序：

```
test=# SELECT * FROM t_test ORDER BY id <-> 100 LIMIT 6;
  id
-----
 100
 101
  99
 102
  98
```

```
  97
(6 rows)
```

如你所见，第一行是一个准确匹配，接下来的匹配就不那么精确并且会变得越来越糟，这个查询将总是返回固定数量的行。

重点是其执行计划：

```
test=# explain SELECT * FROM t_test ORDER BY id <-> 100 LIMIT 6;
                                              QUERY PLAN
-----------------------------------------------------------------------
-----------------
 Limit  (cost=0.28..0.64 rows=6 width=8)
   -> Index Only Scan using idx_id on t_test
             (cost=0.28..5968.28 rows=100000 width=8)
        Order By: (id <-> 100)
(3 rows)
```

如你所见，PostgreSQL 直接就进行了索引扫描，这显著地加快了查询的速度。

在 PostgreSQL 的未来版本中，B-树将很可能也支持 KNN 搜索。在开发邮件列表中已经浮现出了一个增加这种特性的补丁，也许它最终将出现在核心中。让 B-树支持 KNN 特性将最终导致标准数据类型上的 GiST 索引变少。

11.2.4　Dblink–考虑逐步淘汰

使用数据库链接的愿望已经存在了很多年。不过，大约在世纪交替时 PostgreSQL 的外部数据包装器甚至还没有初现端倪，而且也看不到一种传统的数据库链接实现。就在这个时间段，来自加利福尼亚的一位 PostgreSQL 开发者（Joe Conway）通过为 PostgreSQL 引入 dblink 的概念推进了数据库连通性上的工作。虽然 dblink 很好地为人们服务了很多年，但它已经不再是最先进的技术了。

因此，推荐从 dblink 转到更现代化的 SQL/MED 实现（是一种定义外部数据整合到关系数据库中方式的规范）。postgres_fdw 扩展已经在 SQL/MED 之上被开发出来，并且将提供比数据库连通性更多的特性，因为它基本上允许连接到任意数据源。

11.2.5　用 file_fdw 取得文件数据

在一些情况下，有必要从磁盘读取文件并且将其内容披露给 PostgreSQL 作为表。这一点正好可以使用 file_fdw 扩展实现，其思想是提供一个模块允许用户从磁盘读取数据并且使用 SQL 对其进行查询。

首先得安装该模块：

```
CREATE EXTENSION file_fdw;
```

接下来创建一台虚拟服务器：

```
CREATE SERVER file_server FOREIGN DATA WRAPPER file_fdw;
```

file_server 基于 file_fdw 扩展的外部数据包装器，它会告诉 PostgreSQL 如何访问文件。要把一个文件作为表显示，可以使用下面的命令：

```
CREATE FOREIGN TABLE t_passwd
(
     username        text,
     passwd          text,
     uid             int,
     gid             int,
     gecos           text,
     dir             text,
     shell           text
) SERVER file_server
OPTIONS (format 'text', filename '/etc/passwd', header 'false', delimiter':');
```

在笔者的例子中，/etc/passwd 文件将被披露出来。所有的域都必须被列出并且数据类型必须被相应地映射好，所有额外的重要信息用选项传递给该模块。在这个例子中，PostgreSQL 必须了解文件类型（text）、文件的名称、文件的路径还有文件内容使用的定界符。还可以告诉 PostgreSQL 文件中是否有头部。如果该设置为真，第一行就是不重要的，将被跳过。如果用户正好在装载一个 CSV 文件，跳过头部的功能就特别重要。

一旦表被创建，就可以读取数据：

```
SELECT * FROM t_passwd;
```

不出意料，PostgreSQL 返回了/etc/passwd 的内容：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM t_passwd LIMIT 1;
-[ RECORD 1 ]-------
username | root
passwd   | x
uid      | 0
gid      | 0
```

```
gecos     | root
dir       | /root
shell     | /bin/bash
```

在查看其执行计划时，读者将看到 PostgreSQL 使用了外部扫描来从该文件中取得数据：

```
test=# explain (verbose true, analyze true)
     SELECT * FROM t_passwd;
                                                  QUERY PLAN
-----------------------------------------------------------------------
----------------------------------
 Foreign Scan on public.t_passwd (cost=0.00..2.80 rows=18 width=168)
             (actual time=0.022..0.072 rows=61 loops=1)
   Output: username, passwd, uid, gid, gecos, dir, shell
   Foreign File: /etc/passwd
   Foreign File Size: 3484
 Planning time: 0.058 ms
 Execution time: 0.138 ms
(6 rows)
```

这份执行计划还告诉我们文件的尺寸等信息。由于我们在谈论规划器，有一条附注值得一提：PostgreSQL 甚至将会为该文件取得统计信息。规划器会检查文件尺寸并且会为该文件设定与同尺寸的普通 PostgreSQL 表相同的代价。

11.2.6　使用 pageinspect 检查存储

如果用户碰到存储损坏或者可能与表中损坏数据块相关的其他存储问题，pageinspect 扩展可能是用户所需要的模块：

```
test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

pageinspect 的想法是向用户提供一个允许在二进制级别检查表的模块。

在使用该模块时，最重要的事情是取得一块：

```
test=# SELECT * FROM get_raw_page('pg_class', 0);
...
```

这个函数将返回单个块。在前面的例子中，它是 pg_class 参数中的第一块，pg_class 是一个系统表（当然，可以使用任何想要的其他表）。

下一步，可以抽取页面的头部：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
-[ RECORD 1 ]---------
lsn         | 1/35CAE5B8
checksum    | 0
flags       | 1
lower       | 240
upper       | 1288
special     | 8192
pagesize    | 8192
version     | 4
prune_xid | 606562
```

它已经包含了很多有关该页面的信息。如果用户想要了解更多，可以调用 heap_page_items 函数，它会仔细分析页面，并且为其中的每个元组返回一行：

```
test=# SELECT *
     FROM    heap page items(get raw page('pg class', 0))
     LIMIT 1;
-[ RECORD 1 ]---
lp          | 1
lp_off      | 49
lp_flags    | 2
lp_len      | 0
t_xmin      |
t_xmax      |
t_field3    |
t_ctid      |
t_infomask2 |
t_infomask  |
t_hoff      |
t_bits      |
t oid       |
t_data      | ...
```

用户还可以把数据分成多个元组：

```
test=# SELECT tuple data split('pg class'::regclass,
                              t_data, t_infomask, t_infomask2, t_bits)
     FROM heap_page_items(get_raw_page('pg_class', 0))
```

```
     LIMIT 2;
-[ RECORD 1 ]----+--------------------------------
tuple_data_split |
-[ RECORD 2 ]----+--------------------------------
tuple_data_split |
{"\\x6100000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000","\\x98080000","
\\x50ac0c00","\\x00000000","\\x01400000","\\x00000000","\\x4eac0c00","\\x00
000000","\\xbb010000","\\x0050c347","\\x00000000","\\x00000000","\\x01","\\
x00","\\x70","\\x72","\\x0100","\\x0000","\\x00","\\x00","\\x00","\\x00","\
\x00","\\x00","\\x00","\\x01","\\x64","\\xc3400900","\\x01000000",NULL,NULL
}
```

为了阅读这些数据库，用户必须熟悉 PostgreSQL 的磁盘格式；否则，数据可能会显得相当晦涩。

pageinspect 为所有访问方法（表、索引等）提供了函数并且允许详细地分析存储。

11.2.7　用 pg_buffercache 研究缓冲

在简单介绍 pageinspect 扩展之后，笔者想把读者的注意力转到 pg_buffercache 扩展上，它允许用户深入地查看其 I/O 缓冲的内容：

```
test=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

pg_buffercache 扩展为用户提供了一个包含若干域的视图：

```
test=# \d pg_buffercache
      View "public.pg_buffercache"
      Column       |    Type    | Modifiers
------------------+----------+-----------
 bufferid          | integer   |
 relfilenode       | oid       |
 reltablespace     | oid       |
 reldatabase       | oid       |
 relforknumber     | smallint |
 relblocknumber    | bigint    |
 isdirty           | boolean   |
 usagecount        | smallint |
 pinning_backends | integer   |
```

bufferid 域只是一个数字，它标识缓冲区。然后是 relfilenode 域，它指向磁盘上的文件。如果用户想要查看一个文件属于哪个表，可以在 pg_class 系统表中查找，其中也含有一个 relfilenode 域。接下来还有 reldatabase 和 reltablespace 域。注意所有的域都被定义为 oid 类型，因此为了能以更有用的方式提取数据，有必要把系统表连接起来。

relforknumber 域告诉我们被缓冲的是该表的哪一部分，它可能是堆、空闲空间映射或者可见性映射之类的某种其他部件。未来肯定将会有更多类型的关系分支。

接下来的 relblocknumber 告诉我们哪个块被缓冲。最后有 isdirty 等 3 个标志，它们指示块已经被修改、使用计数器以及对块加 pin 的后端数量。

如果读者想要弄明白 pg_buffercache 扩展，有必要增加额外的信息。假定用户想要弄清哪个数据库使用缓冲最多，下面的查询可以帮忙：

```
test=# SELECT datname, count(*),
                      count(*) FILTER (WHERE isdirty = true) AS dirty
     FROM    pg_buffercache AS b, pg_database AS d
     WHERE   d.oid = b.reldatabase
     GROUP BY ROLLUP (1);
  datname | count | dirty
----------+-------+-------
 abc      | 132   | 1
 postgres | 30    | 0
 test     | 11975 | 53
          | 12137 | 54
(4 rows)
```

在这种情况下，必须连接 pg_database 系统表。如你所见，oid 是连接条件，这对于 PostgreSQL 的新手来说可能不是那么明显。

有时用户可能想要知道所连接的数据库中哪些块被缓冲：

```
test=# SELECT relname, relkind,
     count(*),
     count(*) FILTER (WHERE isdirty = true) AS dirty
FROM    pg_buffercache AS b, pg_database AS d, pg_class AS c
WHERE   d.oid = b.reldatabase
        AND c.relfilenode = b.relfilenode
        AND datname = 'test'
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 7;
```

```
           relname           | relkind | count  | dirty
-----------------------------+---------+--------+-------
 t_bloom                     | r       | 8338   | 0
 idx_bloom                   | i       | 1962   | 0
 idx_id                      | i       | 549    | 0
 t_test                      | r       | 445    | 0
 pg_statistic                | r       | 90     | 0
 pg_depend                   | r       | 60     | 0
 pg_depend_reference_index   | i       | 34     | 0
(7 rows)
```

这种情况下，笔者过滤出当前数据库并且与 pg_class 系统表连接，后者含有对象的列表。relkind 列特别值得注意：r 表示表（关系）而 i 表示索引，它将告诉用户正在查看哪种对象。

11.2.8　用 pgcrypto 加密数据

在整个 contrib 模块小节中最强大的模块之一就是 pgcrypto，它最初由 Skype 的一位系统管理员编写并且提供了无数的函数来加密和解密数据。

pgcrypto 为对称以及非对称加密提供了函数。由于其中有大量的函数，笔者推荐查看其文档页面：https://www.postgresql.org/docs/current/static/pgcrypto.html。

由于本章的篇幅有限，所以不可能深入 pgcrypto 模块的所有细节。

11.2.9　用 pg_prewarm 预热缓冲

在 PostgreSQL 正常运作时，它会尝试缓冲重要的数据。shared_buffers 变量很重要，因为它定义了 PostgreSQL 所管理的缓冲的大小。现在的问题是，如果用户重新启动数据库服务器，所有由 PostgreSQL 管理的缓冲内容都将丢失。可能操作系统还将有一些数据以减少对磁盘的影响，但是在很多情况下这并不够。

对这个问题的解决方案被称作 pg_prewarm 扩展：

```
test=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
```

该扩展部署一个函数，它允许我们在需要时显式地预热缓冲：

```
test=# \x
Expanded display is on.
test=# \df *prewa*
List of functions
```

```
-[ RECORD 1 ]-------+-----------------------------------------------
--
Schema              | public
Name                | pg_prewarm
Result data type    | bigint
Argument data types | regclass, mode text DEFAULT 'buffer'::text,
                        fork text DEFAULT 'main'::text,
                        first_block bigint DEFAULT NULL::bigint,
                        last_block bigint DEFAULT NULL::bigint
Type                | normal
```

调用 pg_prewarm 扩展最简单且最常用的方式是要求它缓冲一整个对象：

```
test=# SELECT pg_prewarm('t_test');
 pg_prewarm
------------
        443
(1 row)
```

注意如果表太大导致无法完全放入缓冲，则只有表的一部分会留在缓冲中，不过在大部分情况下这样也够用了。

该函数返回被函数调用处理的 8000 字节块的数量。

如果用户不想缓冲一个对象的所有块，用户还可以选择表中一个指定的范围。在接下来的例子中，读者可以看到在主分支中 10～30 号块被缓冲：

```
test=# SELECT pg_prewarm('t_test', 'buffer', 'main', 10, 30);
pg_prewarm
------------
21
(1 row)
```

如你所见，有 21 个块被缓冲。

11.2.10　用 pg_stat_statements 检查性能

pg_stat_statements 是 contrib 模块中最重要的模块，应该总是被启用，它能够为用户提供更好的性能数据。如果没有 pg_stat_statements 模块，确实很难追踪性能问题。

由于其重要性，pg_stat_statements 已经在本书更早的部分讨论过了。

11.2.11　用 pgstattuple 检查存储

有时候会发生 PostgreSQL 中的表过分增长的情况，表过分增长的技术术语是**表膨**

胀。现在出现的问题是，哪些表发生了膨胀以及有多大程度的膨胀？pgstattuple 扩展将帮助回答这些问题：

```
test=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

这个模块同样会部署若干函数。在 pgstattuple 扩展中，那些函数返回一个由一种组合类型构成的行。因此，这类函数必须在 FROM 子句中调用以确保得到可阅读的结果：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgstattuple('t_test');
-[ RECORD 1 ]------+--------
table_len          | 3629056
tuple_count        | 100000
tuple_len          | 2800000
tuple_percent      | 77.16
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16652
free_percent       | 0.46
```

在笔者的例子中，用于测试的表看起来处于一种相当不错的状态，该表的大小是 3.6MB 并且不包含任何死亡的行，空闲空间也有限。如果用户正在受到表膨胀的折磨，死亡行的数量和空闲空间量就已经不成比例地增长。有一点空闲空间和少量的死亡行是正常的——但是，如果表已经过分增长，那么它大部分会由死亡行和空闲空间构成，需要采取果断的行动控制住局面。

pgstattuple 扩展也提供了函数来检查索引：

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
```

pgstattindex 函数返回很多有关想要观察的索引的信息：

```
test=# SELECT * FROM pgstatindex('idx_id');
-[ RECORD 1 ]------+--------
version            | 2
tree_level         | 1
index_size         | 2260992
root_block_no      | 3
```

```
internal_pages      | 1
leaf_pages          | 274
empty_pages         | 0
deleted_pages       | 0
avg_leaf_density    | 89.83
leaf_fragmentation | 0
```

我们的索引相当密集（89%）。这是一种好现象。索引的默认 FILLFACTOR 设置是 90%，因此接近于 90%的值都表示该索引处于非常不错的状态。

有时候用户不想检查单个表而是检查一个方案中的所有表。如何实现这种要求？一般来说，用户想要处理的对象列表应该在 FROM 子句中。不过在笔者的例子中，函数已经在 FROM 子句中，那么我们怎么让 PostgreSQL 在表的列表上循环呢？答案是用 LATERAL 连接。例如：

```
test=# \x
Expanded display is on.
test=# SELECT tablename, (x).*
     FROM pg_tables,
          LATERAL (SELECT *
                FROM pgstattuple(tablename)) AS x
     WHERE schemaname = 'public';
-[ RECORD 1 ]------+-----------
tablename          | t_ort
table_len          | 114688
tuple_count        | 2354
tuple_len          | 88686
tuple_percent      | 77.33
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 7732
free_percent       | 6.74
```

FROM 子句的第一部分找到我们要查看的表，每一个被返回的表接着被送到 LATERAL 连接。每一个 LATERAL 连接都可以被视作一个 for each 语句。

记住 pgstattuple 必须读取整个对象。如果用户的数据库很大，它就需要很长的时间进行处理。因此，将已经看到的查询结果保存下来是一种不错的做法，这样就可以仔细地观察结果而无须反复地运行这种查询。

11.2.12　用 pg_trgm 进行模糊搜索

pg_trgm 是一个允许用户执行模糊搜索的模块，该模块已经在第 3 章讨论过。

11.2.13　使用 postgres_fdw 连接到远程服务器

数据并不总是只在一个位置上，数据多半散布在基础设施的各处，而且有可能发生需要将位于多个位置的数据集成起来的需求。这一问题的解决方案是由 SQL/MED 标准定义的外部数据包装器。

在本节中将讨论 postgres_fdw 扩展，它是一个允许用户动态从一个 PostgreSQL 数据源获取数据的模块。第一件事情是部署外部数据包装器：

```
test=# \h CREATE FOREIGN DATA WRAPPER
Command: CREATE FOREIGN DATA WRAPPER
Description: define a new foreign-data wrapper
Syntax:
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

幸运的是，CREATE FOREIGN DATA WRAPPER 命令被隐藏在扩展的内部，它可以很容易地使用正常的方式安装：

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

下一步必须定义一台虚拟服务器，它将指向其他主机并且告诉 PostgreSQL 从哪里得到数据。在这些数据的末尾，PostgreSQL 还必须构建一个完整连接字符串——这些服务器数据是 PostgreSQL 必须了解的第一类信息。

接下来将增加用户信息。服务器将只包含主机、端口等信息：

```
test=# \h CREATE SERVER
Command:     CREATE SERVER
Description: define a new foreign server
Syntax:
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version'
]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

为了便于展示，笔者在同一台主机上创建了第二个数据库并且创建了一个表[①]：

```
[hs@zenbook ~]$ createdb customer
[hs@zenbook ~]$ psql customer
customer=# CREATE TABLE t_customer (id int, name text);
CREATE TABLE
customer=# CREATE TABLE t_company (country text, name text, active text);
CREATE TABLE
customer=# \d
          List of relations
 Schema |    Name    | Type  | Owner
--------+------------+-------+-------
 public | t_company  | table | hs
 public | t_customer | table | hs
(2 rows)
```

现在应该把服务器加入标准的 test 数据库中：

```
test=# CREATE SERVER customer_server
     FOREIGN DATA WRAPPER postgres_fdw
     OPTIONS (host 'localhost', dbname 'customer', port '5432');
CREATE SERVER
```

注意所有重要的信息都被存储为 OPTIONS 子句。这还是挺重要的，因为它为用户提供了很多灵活性，不同的外部数据包装器将需要不同的选项。

一旦服务器已经定义好，就应该进行用户映射。如果使用者从一台服务器连接到其他服务器，使用者在两个位置上可能是不同的用户。因此，外部数据包装器要求人们定义实际的用户映射：

```
test=# \h CREATE USER MAPPING
Command:     CREATE USER MAPPING
Description: define a new mapping of a user to a foreign server
Syntax:
CREATE USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

这种语法相当简单并且很容易使用：

[①] 原文是“一台服务器”，根据对应的代码判断，这里应该是笔误。——译者

```
test=# CREATE USER MAPPING FOR CURRENT_USER
     SERVER customer_server
     OPTIONS (user 'hs', password 'abc');
CREATE USER MAPPING
```

再一次，所有重要的信息都隐藏在 OPTIONS 子句中。根据外部数据包装器的类型，选项的列表也将不同。

一旦基础设施就位，就可以创建外部表。创建外部表的语法很像创建普通本地表的语法。所有的列都必须被列出，包括它们的数据类型：

```
test=# CREATE FOREIGN TABLE f_customer
     (
              id       int,
              name     text
     )
SERVER customer_server
OPTIONS (schema_name 'public', table_name 't_customer');
CREATE FOREIGN TABLE
```

所有列就像在普通的 CREATE TABLE 子句中那样被列出，特别之处在于外部表指向一个位于远端的表。方案的名称和远端表的名称必须在 OPTIONS 子句中被指定。

外部表被创建好后就可以使用了：

```
test=# SELECT * FROM f_customer ;
 id | name
----+------
(0 rows)
```

要查看 PostgreSQL 内部做的事情，运行带有 analyze 参数的 EXPLAIN 子句是一种好办法。它将揭示服务器中到底在做些什么：

```
test=# EXPLAIN (analyze true, verbose true)
     SELECT * FROM f_customer ;
                                           QUERY PLAN
-----------------------------------------------------------------------
---------------------
 Foreign Scan on public.f_customer
     (cost=100.00..150.95 rows=1365 width=36)
     (actual time=0.221..0.221 rows=0 loops=1)
   Output: id, name
   Remote SQL: SELECT id, name FROM public.t_customer
```

```
 Planning time: 0.067 ms
 Execution time: 0.451 ms
(5 rows)
```

这里很重要的部分是 Remote SQL。外部数据包装器将发送一个查询到另一端并且取得尽可能小的数据。在远端会执行尽可能多的限制以确保不会有太多数据需要被拿回到本地处理。过滤条件、连接甚至聚集都可以被在远程执行（从 PostgreSQL 10.0 开始）。

虽然 CREATE FOREIGN TABLE 子句确实是一种很好用的东西，但是一次又一次地列出所有那些列也很麻烦。

这个问题的解决方案被称为 IMPORT 子句。它允许用户快速简便地把整个方案导入用户的本地数据库中并且创建外部表：

```
test=# \h IMPORT
Command:     IMPORT FOREIGN SCHEMA
Description: import table definitions from a foreign server
Syntax:
IMPORT FOREIGN SCHEMA remote_schema
    [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
    FROM SERVER server_name
    INTO local_schema
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

IMPORT 允许用户简便地链接到大型的表集合。它还能减少出现拼写和输入错误的概率，因为所有的信息都直接从远程数据源取得。

```
test=# IMPORT FOREIGN SCHEMA public
     FROM SERVER customer_server
     INTO public;
IMPORT FOREIGN SCHEMA
```

在这个例子中，所有以前在 public 模式中创建的表都被直接链接进来。如你所见，现在所有的远程表都可用了：

```
test=# \det
        List of foreign tables
 Schema |   Table     |      Server
--------+-------------+-----------------
 public | f_customer | customer_server
 public | t_company  | customer_server
 public | t_customer | customer_server
(3 rows)
```

- 处理错误

创建外部表实际并不难——但是，有时候人们会犯错，或者口令被更改了。为了处理这类问题，PostgreSQL 提供了两个命令。

ALTER SERVER 允许用户修改服务器：

```
test=# \h ALTER SERVER
Command:     ALTER SERVER
Description: change the definition of a foreign server
Syntax:
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

用户可以使用这个命令为一台特定的服务器增加以及移除选项，在用户遗忘了某些事情的情况下这个命令很有用。

如果使用者想要修改用户信息，使用者也可以修改用户映射：

```
test=# \h ALTER USER MAPPING
Command:     ALTER USER MAPPING
Description: change the definition of a user mapping
Syntax:
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER |
PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

SQL/MED 接口会被定期地改进，并且就在读者阅读本书时新的特性也在加入。在未来，更多地优化将被放入核心中，使得 SQL/MED 接口成为改进可扩展性的一种很好的选择。

11.3　其他有用的扩展

到目前为止，所描述的扩展都是 PostgreSQL 的 contrib 包的组成部分。但是，读者已经看到的包并非 PostgreSQL 社区中唯一可用的包，还有更多包允许使用者做各种各样的事情。

可惜本章的篇幅太短，我们没有办法深入所有扩展中。模块的数量每天都在增加，所以本书也不可能涵盖到所有的内容。因此，笔者只想点出自己认为最重要的那些扩展。

PostGIS（http://postgis.net/）是开源世界中的地理信息系统（GIS）数据库接口，它已经在全球很多应用中被采用并且是关系型开源数据库世界中的事实标准，它是一种专业而且极强大的解决方案。

如果使用者正在查找地理空间路径，pgRouting 就是使用者可能在寻找的工具。它提供了多种算法来寻找位置之间的最佳连接并且可以在 PostgreSQL 之上工作。

在本章中，读者已经学到了 postgres_fdw 扩展，它允许使用者连接到某个其他的 PostgreSQL 数据库。还有更多的外部数据包装器存在，其中一个最著名且最专业的是 oracle_fdw 扩展。它允许使用者与 Oracle 进行集成并且通过网络从 Oracle 中取得数据，就好像用 postgres_fdw 扩展所作的事情那样。

11.4　总　　结

在本章中，读者学到了一些最有前途的包含在 PostgreSQL 标准发布中的模块。这些模块非常多样，包括了从数据库连通性到大小写不敏感文本以及检查服务器的模块。在介绍完扩展之后，笔者将会把读者注意力转移到迁移上。读者将学到如何以最简单的方式转移到 PostgreSQL 上。

第 12 章　在 PostgreSQL 中排查错误

在本书的第 11 章中，读者学到了一些被广泛采用的扩展，它们对用户的部署是有力的促进。接下来，本书将向读者介绍 PostgreSQL 的故障排除。主要想法是给读者一种系统的方法检查并且修复其系统。

本章将关注下列主题：

- 处理未知的数据库。
- 获得一份简要概述。
- 确定关键的瓶颈。
- 处理存储损坏。
- 检查坏掉的复制系统。

要记住很多事情都可能会出错，因此有必要专业地监控数据库。

12.1　着手处理一个陌生的数据库

如果用户恰好管理着一个大规模系统，他/她可能不知道系统到底在做什么。管理数百个系统则意味着用户将无法了解在每个系统中正在发生的事情。

在排查故障时，最重要的事情可以归结成一个词——数据。如果没有足够的数据，就没有办法修复问题。因此，排查故障的第一步总是设置一种 pgwatch2 之类的监控工具，它们能让用户洞悉其数据库服务器内部。

一旦从报告中发现值得检查的情况，就意味着这种有条理地着手处理该系统的方式发挥了作用。

12.2　检查 pg_stat_activity

笔者推荐的第一件事情是检查 pg_stat_activity[①]。回答下列问题：

- 当前有多少并发事务在你的系统中运行？
- 是否总是在 query 列中看到类似的查询类型？
- 有没有看到已经运行了很长时间的查询？

[①] 原文是“pg_stat_statements”，但根据上下文的意思，应该是笔误，正确应为“pg_stat_activity”。——译者

- 有没有没有被授予的锁？
- 是否看到来自可疑主机的连接？

应该总是首先检查 pg_stat_activity 视图，因为它反映了系统中正在发生什么事情。当然，图形化监控也能让用户对系统形成第一印象。但是，最终还是要归结到实际运行在服务器上的查询。因此，由 pg_stat_activity 提供的好的系统概述比查出问题更加关键。

为了便于读者使用，笔者已经编写了若干查询，笔者认为它们有助于尽快找到问题。

- 查询 pg_stat_activity

下面的查询展示在数据库中当前正在执行多少查询：

```
test=# SELECT datname,
        count(*) AS open,
        count(*) FILTER (WHERE state = 'active') AS active,
        count(*) FILTER (WHERE state = 'idle') AS idle,
        count(*) FILTER (WHERE state = 'idle in transaction') AS
idle in trans
FROM pg_stat_activity
GROUP BY ROLLUP(1);
 datname | open | active | idle | idle_in_trans
---------+------+--------+------+---------------
 dev     |    1 |      1 |    0 |             0
 test    |    2 |      1 |    0 |             1
         |    3 |      2 |    0 |             1
(3 rows)
```

为了在同一屏上显示尽可能多的信息，这个查询中使用了部分聚集。读者可以看到 active、idle 以及 idle in transaction 的查询。如果看到大量“idle in transaction”查询，绝对有必要深究一下那些事务已经打开了多久：

```
test=# SELECT pid, xact_start, now() - xact_start AS duration
   FROM pg_stat_activity
   WHERE state LIKE '%transaction%'
   ORDER BY 3 DESC;
  pid  |          xact_start           |    duration
-------+-------------------------------+-----------------
 22503 | 2017-03-15 13:13:08.368974+01 | 22:14:12.126463
(1 row)
```

列表中的事务已经被打开了超过 22 小时。现在的主要问题是，为什么一个事务可以被打开那么久？在大部分应用中，耗费如此长时间的事务非常可疑并且有可能极度危

险。然而，危险来自何处？正如读者在本书中已经了解的，VACUUM 子句只能清理那些不再对任一事务可见的死亡行。现在，如果一个事务保持打开状态数小时甚至数天，VACUUM 子句就无法产生有用的结果，最终将会导致表膨胀。

因此，笔者高度推荐确保长事务都处于监控之下，或者在运行太久的情况下杀死它们。从版本 9.6 开始，PostgreSQL 有了一种称为“**snapshot too old**”的特性，这允许用户在快照存在过久的情况下中止长事务。

还有一种好的方法是检查是否有长查询运行：

```
test=# SELECT now() - query start AS duration, datname, query
   FROM pg_stat_activity
   WHERE state = 'active'
   ORDER BY 1 DESC;
    duration      | datname | query
-----------------+---------+------------------------------------
 00:00:38.814526 | dev     | SELECT pg sleep(10000);
 00:00:00        | test    | SELECT now() - query_start AS duration,
                             datname, query FROM pg_stat_activity WHERE
                             state = 'active' ORDER BY 1 DESC;
(2 rows)
```

在这种情况下，会得到所有的活动查询，并且该语句会计算一个查询已经活动了多久。用户经常会看到相似的查询出现在顶部，它们能给出一些有关于系统中正在发生什么的线索。

1. 处理 Hibernate 语句

很多 ORM（例如 Hibernate）会产生疯狂的长 SQL 语句。这会带来一个小问题，pg_stat_activity 将只在系统视图中存放查询的前 1024 字节，剩下的部分都会被截去。对于 Hibernate 之类 ORM 生成的长查询来说，查询可能在有趣的部分（FROM 子句等）开始之前就被切断了。

这一问题的解决方案是在 postgresql.conf 文件中设置一个 config 参数：

```
 test=# SHOW track_activity_query_size;
 track_activity_query_size
---------------------------
 1024
(1 row)
```

将这个参数调高到合理的值（可能是 32768）并且重启 PostgreSQL。用户将能看到

长很多的查询并且能够更容易地检测出问题。

2. 确定查询的来源

在观察 pg_stat_activity 时，有一些域将会告诉用户查询来自哪里：

```
client_addr       | inet                     |
client_hostname   | text                     |
client_port       | integer                  |
```

那些域将包含 IP 地址和主机名（如果配置）。但是，如果所有的应用都位于同一台应用服务器上，就会导致它们都从完全相同的 IP 发送其查询，那么会发生什么？用户很难知道一个特定的查询到底由哪个应用生成。

这类问题的解决方案是要求开发者设置一个 application_name 变量：

```
test=# SHOW application_name ;
 application_name
------------------
 psql
(1 row)

test=# SET application_name TO 'some_name';
SET
test=# SHOW application_name ;
 application_name
------------------
 some_name
(1 row)
```

如果人们都合作，application_name 变量将会出现在该系统视图中，这样就更容易看到查询从哪里来。application_name 变量也可以被作为连接字符串的一部分进行设置。

12.3　检查慢查询

在检查了 pg_stat_activity 之后，有必要看一看慢的、很消耗时间的查询。基本上有两种方法来解决该问题：

- 在日志中查找个别的慢查询。
- 查找花费太多时间的查询类型。

寻找单个慢查询是经典的性能调优方法。通过设置 log_min_duration_statement 变量

为一个期望的阈值，PostgreSQL 将开始为每个超过该阈值的查询在日志中记下一行。慢查询日志默认被关闭：

```
test=# SHOW log_min_duration_statement;
 log_min_duration_statement
----------------------------
 -1
(1 row)
```

不过，把这个变量设置为一个合理的值非常重要。根据用户的负载，期望的时间当然会发生变化。

在很多情况下，每个数据库的这个期望值可能都不同。因此，也可以以更细的粒度使用这个变量：

```
test=# ALTER DATABASE test
   SET log_min_duration_statement TO 10000;
ALTER DATABASE
```

如果用户的数据库面临着不同的负载，那么只为特定的数据库设置这个参数就非常有必要。

在使用慢查询日志时，有必要考虑一个重要的因素——很多较小的查询可能导致比一小部分慢查询更多的负载。当然，察觉到慢查询个体总是有必要的，但是有时候这些查询并不是问题。考虑下面的例子：在用户的系统上，1 百万个花费 500 毫秒的查询与一些运行数分钟的分析查询一起执行。显然，实际的问题将不会出现在慢查询日志中，而每一次数据导出、每一次索引创建以及每一次批量装载（这些工作在大部分情况下都无法避免）将会在日志中经常出现并且把用户导向错误的方向。

因此，笔者的个人建议是慢查询日志有必要使用，但是要小心地使用。而且最重要的是，要意识到通过慢查询日志到底要发现什么。

在笔者看来，更好的方法是更紧密地结合 pg_stat_statements 变量的使用。它将提供聚合信息而不仅是单个查询的信息。在本书稍早的部分已经讨论过 pg_stat_statements 变量，但该模块的重要性再怎么强调也不为过。

12.3.1　检查个体查询

有时，可以确定慢查询，但用户仍然没有有关问题真相的线索。当然，下一步是检查该查询的执行计划并且看看发生了什么。要在计划中找出对糟糕运行时间负有责任的那些关键操作相当简单。尝试使用下面的检查表：

- 尝试查看计划中什么地方时间开始飞涨。
- 检查缺失的索引（糟糕性能的主要原因之一）。
- 使用 EXPLAIN 子句（buffers true、analyze true 等）来看看查询是否用了太多缓冲区。
- 打开 track_io_timing 参数以找出是否有 I/O 问题或者 CPU 问题（明确检查是否有随机 I/O 进行）。
- 查找错误的估计并且尝试修复。
- 查找被过于频繁执行的存储过程。
- 如果有这样的存储过程，看看其中是否有一些可以被标记为 STABLE 或者 IMMUTABLE。

注意，pg_stat_statements 没有把解析时间计算在内，因此如果用户的查询非常长（查询字符串），pg_stat_statements 可能会有一点误导性。

12.3.2　用 perf 深入研究

在大部分情况下，做完这个小的检查表将帮助用户以非常快速且有效的方式查出主要的问题。但是，即便从数据库引擎提取的信息有时候也不够。

perf 是一种用于 Linux 的分析工具，它允许用户直接查看哪些 C 函数导致了用户系统上的问题。通常 perf 默认没有被安装，因此推荐安装它。要在服务器上使用 perf，只需要以 root 权限登录并且运行：

```
perf top
```

每隔若干秒屏幕将会自动刷新，用户将有机会可以看到有哪些活动正在进行。接下来的列表展示了一个标准的只读测试基准的模样：

```
Samples: 164K of event 'cycles:ppp', Event count (approx.): 109789128766
Overhead  Shared Object             Symbol
   3.10%  postgres                  [.] AllocSetAlloc
   1.99%  postgres                  [.] SearchCatCache
   1.51%  postgres                  [.] base_yyparse
   1.42%  postgres                  [.] hash_search_with_hash_value
   1.27%  libc-2.22.so              [.] vfprintf
   1.13%  libc-2.22.so              [.] _int_malloc
   0.87%  postgres                  [.] palloc
   0.74%  postgres                  [.] MemoryContextAllocZeroAligned
   0.66%  libc-2.22.so              [.] __strcmp_sse2_unaligned
   0.66%  [kernel]                  [k] _raw_spin_lock_irqsave
```

```
    0.66%  postgres                    [.] _bt_compare
    0.63%  [kernel]                    [k] __fget_light
    0.62%  libc-2.22.so                [.] strlen
```

可以看到在我们的例子中没有单个函数占用过多的 CPU 时间，这告诉我们系统状态挺好。

但情况并非总是这样。有一种非常常见的问题，被称为自旋锁竞争。那是什么？自旋锁（https://en.wikipedia.org/wiki/Spinlock）被 PostgreSQL 核心用来同步诸如缓冲区访问之类的事情。自旋锁是现代 CPU 提供的一种特性，它被用来避免操作系统对于小型操作（例如增加一个数字）的交互作用。它是一种好东西，但是在一些非常特殊的情况下，自旋锁可能变得很古怪。如果用户正面临着自旋锁竞争，其现象如下：

- 非常高的 CPU 负载。
- 难以置信的低吞吐（通常花费数毫秒的查询突然需要花费数秒）。
- I/O 通常很低，因为 CPU 在忙于交易锁。

在很多情况下，自旋锁竞争都是突然发生的。用户的系统本来好好的，突然负载暴增而吞吐直线下降。perf top 命令将会显示大部分的时间被花费在一个名为 s_lock 的 C 函数中。如果情况就是这样，用户应该尝试下面的办法：

```
huge_pages = try                # on, off, or try
```

将 huge_pages 从 try 改成 off。在操作系统级别也一同关闭大页特性可能是一个好主意。通常，似乎一些内核比其他的内核更倾向于产生这类问题。Red Hat 2.6.32 系列似乎尤其糟糕（注意笔者用的是“似乎”）。

如果用户在使用 PostGIS，perf 也非常有吸引力。如果列表中顶部的函数都是 GIS 相关的（一些底层库），用户应该认识到问题不太可能来自于不好的 PostgreSQL 调优，只是因为执行了需要花费很多时间完成的昂贵操作而已。

12.4　检 查 日 志

如果系统出现了问题的征兆，有必要检查日志看看发生了什么。这里的重点是，并非所有的日志项都是被平等创建的。PostgreSQL 采用了一种日志项的层级结构，其范围从 DEBUG 消息一直到 PANIC 消息。

对于管理员来说，下面 3 种错误级别的重要性很高：

- ERROR
- FATAL
- PANIC

ERROR 被用于语法错误、权限相关错误等问题。用户的日志将总是包含错误消息。关键问题是，一种特定错误类型出现的频率如何？产生上百万的语法错误显然不是运行数据库服务器的理想方式。

FATAL 比 ERROR 更加可怕，例如无法为共享内存名称分配内存时或者出现意外的 walreceiver 状态时都将看到 FATAL 消息。换种说法，这类错误消息已经极其吓人了，它们将告诉用户系统中出现了问题。

最后是 PANIC。如果用户碰到了这种消息，那么就说明数据库系统真出现了问题。PANIC 的典型例子就是锁表损坏或者创建了太多信号量。这种消息将会导致停机。

12.5　检查缺失的索引

一旦做完了前 3 个步骤，就有必要从总体上来看看系统的性能。正如笔者在本书中反复提到的，索引缺失是造成性能不好非常重要的原因。因此，只要碰到一个很慢的系统，推荐检查有没有索引缺失并且部署所需要的索引。

通常客户会要求我们优化 RAID 级别、调优内核或者做其他一些花俏的事情。实际上，那些复杂的请求常常会归结为少量缺失的索引。笔者认为，总是有必要花一些额外的时间来检查是否想要的索引都已经被创建。检查索引缺失既不困难也花不了多少时间，所以不管面临着何种性能问题，都应该做这种检查。

下面是笔者最喜欢的查询，它可以让我们形成哪里可能缺少索引的印象：

```
SELECT   schemaname, relname, seq_scan, seq_tup_read,
           idx_scan, seq_tup_read / seq_scan AS avg
FROM     pg_stat_user_tables
WHERE    seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 20;
```

这个查询尝试查找经常被扫描的大型表（avg 高），那些表将出现在结果的顶部。

12.6　检查内存和 I/O

做完缺失索引的查找检查之后，接下来可以检查内存和 I/O。为了弄明白系统中正在进行什么样的工作，有必要激活 track_io_timing。如果它被打开，PostgreSQL 将收集有关磁盘等待的信息并且展示给用户。

客户常常会问的一个主要问题是：如果我们增加更多磁盘，系统是不是能够更快一

些？我们当然可以猜想一下会发生什么，但是通常来说实际的测量才是更好且更有用的策略。启用 track_io_timing 将帮助用户汇集数据以真正地找出结论。

PostgreSQL 以多种方式向用户显露磁盘等待的信息。一种方法是查看 pg_stat_database：

```
test=# d pg_stat_database
           View "pg_catalog.pg_stat_database"
     Column     |           Type           | Modifiers
----------------+--------------------------+-----------
 datid          | oid                      |
 datname        | name                     |
...
 conflicts      | bigint                   |
 temp_files     | bigint                   |
 temp_bytes     | bigint                   |
...
 blk_read_time  | double precision         |
 blk_write_time | double precision         |
```

其中有两个域与上述目的有关：blk_read_time 和 blk_write_time。它们将告诉我们 PostgreSQL 花在等待 OS 响应上的时间。注意这里实际上没有测量磁盘等待，而是测量操作系统返回数据所需的时间。如果操作系统产生了缓冲命中，这种时间将会相当短。如果 OS 真地不得不去做随机 I/O，我们将看到取得单个块甚至就需要若干毫秒。

在很多情况下，当 temp_files 以及 temp_bytes 显示较高的数字时，blk_read_time 和 blk_write_time 值就会较高。还有很多情况下，blk_read_time 和 blk_write_time 的值很高意味着不好的 work_mem 或 maintenance_work_mem 设置。牢记这一点：如果 PostgreSQL 无法在内存中执行操作，它就不得不溢出到磁盘，temp_files 是检测这种情况的方法。只要 temp_files 中出现值，就有可能发生令人不快的磁盘等待。

虽然在每个数据库级别上的全局视图很有意义，但从这里无法得到实际问题来源的深层信息。常见的情况是只有少数的查询需要为不好的性能负责，找到这些查询的方法是使用 pg_stat_statements：

```
test=# d pg stat statements
           View "public.pg_stat_statements"
        Column         |       Type        | Modifiers
-----------------------+-------------------+-----------
...
 query                 | text              |
 calls                 | bigint            |
```

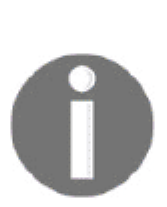
```
 total_time            | double precision |
...
 temp_blks_read        | bigint           |
 temp_blks_written     | bigint           |
 blk_read_time         | double precision |
 blk_write_time        | double precision |
```

用户将可以以每个查询为基础，查看是否有磁盘等待。其中重要的部分是结合 total_time 查看 blk_时间，它们之间的比例才是最有价值的信息。通常，如果一个查询有 30%的时间用于磁盘等待，那么它就可以被视为严重的 I/O 密集型查询。

检查完 PostgreSQL 的系统表，接下来可以检查 Linux 上 vmstat 命令的结果。此外，也可以使用 iostat 命令：

```
[hs@zenbook ~]$ vmstat 2
procs -----------memory--------- -swap- -io- -system-  ------cpu-----
r b   swpd   free   buff cache    si  so bi bo in  cs   us sy id wa st
0 0   367088 199488 96   2320388  0   2  83 96 106 156  16 6  78 0  0
0 0   367088 198140 96   2320504  0   0  0  10 595 2624 3  1  96 0  0
0 0   367088 191448 96   2320964  0   0  0  8  920 2957 8  2  90 0  0
```

在做数据库工作时，用户应该关注 3 个域：bi、bo 和 wa。bi 域告诉我们块读取的次数，1000 等效于“MB/s”。bo 域有关写出的块，它告诉我们被写出到磁盘的数据量。在某种程度上，bi 和 bo 是原始的吞吐量。笔者认为单靠这两个数字不能说明性能受到了伤害，真正的问题是较高的 wa 值。较低的 bi 和 bo 值结合较高的 wa 值将告诉我们有可能出现了磁盘瓶颈，而这种瓶颈很可能与系统中发生的大量随机 I/O 有关。wa 值越高，查询就越慢，因为用户需要等待磁盘响应。



好的原始吞吐是好事，但它也可能表示有问题。如果在一个 OLTP 系统上需要高吞吐，原始吞吐将会告诉用户没有足够的 RAM 来缓冲数据或者 PostgreSQL 由于缺少索引而不得不读取太多数据。记住这些检测数据之间都是相互联系的，不应该孤立地看待数据。

12.7　了解值得注意的错误场景

在给出了追踪最常见问题的基本指导方针后，本节将讨论 PostgreSQL 世界中发生的一些最常见的错误场景。

12.7.1　面对 clog 损坏

PostgreSQL 有一种被称作**提交日志**的东西，它也被叫作 **clog**。它跟踪系统中每一个事务的状态并且帮助 PostgreSQL 判断一行是否可见。通常，一个事务可以处于 4 种状态：

```
#define TRANSACTION_STATUS_IN_PROGRESS    0x00
#define TRANSACTION STATUS COMMITTED      0x01
#define TRANSACTION_STATUS_ABORTED        0x02
#define TRANSACTION_STATUS_SUB_COMMITTED  0x03
```

clog 在 PostgreSQL 数据库实例中有一个单独的目录。

在过去，人们已经报告了一种被称为 **clog 损坏**的问题，这种问题可能由有缺点的磁盘或者 PostgreSQL 中的缺陷（已经在过去的这些年中被修复）造成。碰到损坏的提交日志是非常让人不快的事情，因为所有的数据都还在，但是 PostgreSQL 却再也无法知道哪些可用哪些不可用。这一领域的损坏无异于一场彻头彻尾的灾难。

管理员如何能够得知提交日志损坏呢？管理员通常将会看到：

```
ERROR: could not access status of transaction 118831
```

如果 PostgreSQL 无法访问事务的状态，当然就有麻烦。主要的问题是如何修复这类问题？直截了当地说，没有办法真正地修复该问题——用户只能尝试尽可能多地抢救数据。

如前所述，提交日志为每个事务保持两个位。这意味着在每个字节中存有 4 个事务，因此每个块中有 32768 个事务。一旦找出哪个块损坏，用户就可以伪造事务日志：

```
dd if=/dev/zero of=<data directory location>/pg_clog/0001
   bs=256K count=1
```

用户可以使用 dd 来伪造事务日志并且将提交状态设置为想要的值。其核心问题是，应该使用哪种事务状态？答案是任何状态实际上都是错误的，因为我们实在不知道那些事务是怎样结束的。不过为了让损失的数据更少，通常把那些事务设置为已提交会比较好。在决定怎么做破坏最小时，实际上取决于用户的工作负载和数据。

当用户不得不这样做时，应该尽可能少地伪造 clog。注意，我们实际上是在伪造提交状态，这对数据库引擎来说并不是什么好事。

一旦完成了伪造 clog，用户应该尝试尽快创建一个备份并且从零开始重建数据库实例。用户正在使用的系统不再可信，因此用户应该尝试尽快提取出数据。请记住：用户将要抽取的数据可能会是矛盾的和错误的，因此要确保对能从数据库服务器中抢救出来的数据做一些质量检查。

12.7.2　理解检查点消息

检查点对于数据完整性以及性能都必不可少。检查点之间离得越远，通常性能就越好。在 PostgreSQL 中，默认的配置通常相当保守，因此检查点相对比较快。如果做检查点的同时数据库核心中有大量数据被改变，PostgreSQL 可能会告诉我们它认为检查点过频。日志文件将显示下列项：

```
LOG: checkpoints are occurring too frequently (2 seconds apart)
LOG: checkpoints are occurring too frequently (3 seconds apart)
```

在转储/恢复或者其他大型操作造成的重度写过程中，PostgreSQL 可能会提示配置参数设置得过低。日志中会记录一条消息来告诉我们这些。

如果用户看到这类消息，出于性能原因，笔者强烈推荐通过大幅增加 max_wal_size 参数（较老的版本中该设置被称作 checkpoint_segments）来增加检查点的距离。在最近版本的 PostgreSQL 中，默认的配置已经比以前常用的配置好了很多。不过，过于频繁地写数据仍然很容易发生。

当用户看到一条有关检查点的消息时，有一点必须牢记：检查点过频根本就不危险——它只是会导致不太好的性能。写入确实会慢很多但是数据没有危险。把两个检查点间的距离增加到足够大将会避免这种错误并且同时加快数据库实例的速度。

12.7.3　管理损坏的数据页面

PostgreSQL 是一种非常稳定的数据库系统，它会尽可能地保护数据并且已经在过去的岁月里证明了其价值。但是，PostgreSQL 依赖于硬件和一种正确工作的文件系统。如果存储损坏，PostgreSQL 也将损坏——除了增加复制系统确保万无一失之外确实无法做到更多。

有时候，文件系统或者磁盘可能会失效。但是在很多情况下，整个系统将不会变得越来越糟，只是有若干块由于某种原因损坏。近来，我们已经在虚拟环境中见过这类情况的发生。一些虚拟机不会默认刷入磁盘，这就意味着 PostgreSQL 不能信赖被写入到磁盘的东西。这类行为可能导致很难预测的随机问题。

如果一个块无法再被读取，用户可能会看到下面这样的一条错误消息：

```
"could not read block %u in file "%s": %m"
```

将要运行的查询将会报错并且停止工作。

幸运的是，PostgreSQL 有方法能处理这类事情：

```
test=# SET zero_damaged_pages TO on;
SET
test=# SHOW zero_damaged_pages;
 zero_damaged_pages
--------------------
 on
(1 row)
```

zero_damaged_pages 是一个配置变量，它允许用户处理损坏的页面。PostgreSQL 将拿到这样的块并且将它简单地用零填充，而不是抛出错误。

注意，这无疑将导致数据丢失。但不管怎样，数据之前就已经损坏或者丢失，因此这只是一种处理存储系统中故障导致的数据损坏的方式而已。

笔者会建议大家小心地处理 zero_damaged_pages 变量——在调用它时要意识到自己在做什么。

12.7.4　粗心的连接管理

在 PostgreSQL 中，每一个数据库连接都是一个单独的进程。所有那些进程使用共享内存（在技术上来看，大部分情况下是映射内存，但对这个例子来说没什么区别）同步。这块共享内存包含 I/O 缓冲、活动的数据库连接列表、锁以及让系统功能正确的更多关键数据。

当一个连接被关闭时，它将从共享内存中移除所有相关项并且让系统处于一种健全的状态。但是，当数据库连接由于某种原因崩溃时会发生什么？

postmaster（主进程）将检测到一个子进程不见了。然后，所有其他连接都将被终止并且一个前滚进程将被初始化。为什么要这样？当一个进程崩溃时，它可能正好在编辑共享内存区域。换句话说，一个崩溃的进程可能会在共享内存中留下损坏的状态。因此，postmaster 会行动起来并且在这种损坏在系统中传播之前把所有人都从系统中踢出去。所有的内存会被清理并且所有人都必须重新连接。

从最终用户的角度来看，这种现象感觉就是 PostgreSQL 整个崩溃并且重启，但事实并非如此。由于进程无法在自己崩溃（段错误）或收到某些其他信号时做出反应，因此清除所有东西对于保护数据来说绝对是至关重要的。如果用户在数据库连接上使用 kill -9 命令也会发生同样的事情。该连接无法捕捉到该信号（-9 不能根据定义捕捉），因此 postmaster 必须做出反应。

12.7.5　与表膨胀斗争

在使用 PostgreSQL 时，表膨胀是最重要的问题之一。如果遇到不好的性能，找出是

否有对象使用了比预期更多的空间总是一个好主意。

怎样才能找出哪里发生了表膨胀呢？可以考虑检查 pg_stat_user_tables 参数：

```
test=# d pg_stat_user_tables
           View "pg_catalog.pg_stat_user_tables"
        Column        |           Type           | Modifiers
---------------------+--------------------------+-----------
 relid                | oid                      |
 schemaname           | name                     |
 relname              | name                     |
...
 n live tup           | bigint                   |
 n_dead_tup           | bigint                   |
```

n_live_tup 和 n_dead_tup 域将会让我们对表膨胀的情况形成印象。用户也可以使用第 11 章介绍的 pgstattuple。

如果出现了严重的表膨胀，我们能做什么？首选项是运行 VACUUM FULL 子句。但问题是 VACUUM FULL 子句需要一个表锁。在一个大型的表上，这种加锁的操作可能会是一个大问题，因为在表被重写期间用户无法对该表做写操作。

如果使用的是 PostgreSQL 9.6 以上的版本，用户可以使用一种名为 pg_squeeze 的工具，它会在后台组织表而无须阻塞其他操作：http://www.cybertec.at/introducing-pg_squeeze-a-postgresql-extension-to-auto-rebuild-bloated-tables/。



如果要重新组织一个非常大的表，这种工具特别有用。

12.8　总　　结

在本章中，读者学到了如何系统地着手处理一个数据库系统并且检测使用 PostgreSQL 会遇到的最常见的问题。读者学到了一些重要的系统表以及其他一些重要的因素，它们决定着我们是否能够成功地完成故障排除工作。

在本书的最后一章中，我们将聚焦于向 PostgreSQL 迁移这一话题。如果读者正在使用 Oracle 或者某种其他的数据库系统，读者可能想要试试 PostgreSQL。第 13 章将告诉大家如何做这件事。

第 13 章　迁移到 PostgreSQL

在第 12 章，笔者展示了如何处理与 PostgreSQL 故障排除相关的最常见的问题。重点是用一种系统的方法来追踪问题，这也是笔者想要尝试提供的东西。

本书的最后一章与转移到 PostgreSQL 有关。很多读者可能仍然忍受着商业数据库许可证花销的煎熬，笔者想要告诉所有这些用户一条出路，并且展示如何把数据从某种专有系统中转移到 PostgreSQL。转移到 PostgreSQL 不仅仅在财务方面有意义，还可以为用户带来更多先进的特性以及更多的灵活性。PostgreSQL 有非常多的特性可以提供给用户，并且在我们谈论它时不断有新特性被加入。

本章将涵盖以下内容：

- 把 SQL 语句迁移到 PostgreSQL。
- 从 Oracle 转移到 PostgreSQL。
- 从 MySQL 转移到 PostgreSQL。

在本章结束时，读者应该能够从某种其他系统把一个基本的数据库转移到 PostgreSQL。

13.1　迁移 SQL 语句到 PostgreSQL

在从一个数据库转移到 PostgreSQL 时，有必要先看看哪种数据库引擎提供哪种功能。转移数据和结构本身通常相当容易。但是，重写 SQL 可能就不是这样了。因此，笔者决定用一个小节来介绍 SQL 的多种先进特性以及它们在当今数据库引擎中的可用性。

在本节中将覆盖最重要的内容。为了成功地转移数据库，有必要理解哪种数据库支持哪些特性。在接下来的小节中，笔者选择了最常见的数据库引擎并且给出了它们的对比。

13.1.1　使用侧连接

在 SQL 中，侧连接（lateral join）基本上可以被视为某种循环。它允许我们参数化一个连接并且多次执行 LATERAL 子句中的所有东西。这里是一个简单的例子：

```
test=# SELECT *
   FROM generate_series(1, 4) AS x,
         LATERAL (SELECT array_agg(y)
                  FROM generate_series(1, x) AS y
                  ) AS z;
```

```
 x | array_agg
---+-----------
 1 | {1}
 2 | {1,2}
 3 | {1,2,3}
 4 | {1,2,3,4}
(4 rows)
```

这个例子将为每个 x 调用 LATERAL 子句。对于最终用户来说，这说白了就是某种循环。

- 支持 lateral

在本章中，读者将学到有关多种数据库的很多知识并且弄清哪种引擎支持哪些特性。一种重要的 SQL 特性就是侧连接。下面的列表展示了哪些引擎支持侧连接以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 9.3 开始支持。
- SQLite：不支持。
- DB2 LUW：从版本 9.1（2005）开始支持。
- Oracle：从 12c 开始支持。
- MS SQL Server：从 2005 开始支持，但使用了不同的语法。

13.1.2　使用分组集

如果用户想要同时运行多于一种聚集，分组集就非常有用。使用分组集可以加速聚集，因为它不需要一次又一次地处理数据。

例如：

```
test=# SELECT x % 2, array_agg(x)
         FROM generate series(1, 4) AS x
         GROUP BY ROLLUP (1);
 ?column? | array_agg
----------+-----------
        0 | {2,4}
        1 | {1,3}
          | {2,4,1,3}
(3 rows)
```

PostgreSQL 提供的不止 ROLLUP 子句，还支持 CUBE 和 GROUPING SETS 子句。

- 支持分组集

分组集本质上是在单个查询中生成多于一种聚集。下面的列表展示了哪些引擎支持分组集以及哪些不支持。

- MariaDB：从 5.1 开始只支持 ROLLUP 子句（不完全支持）。
- MySQL：从 5.0 开始只支持 ROLLUP 子句（不完全支持）。
- PostgreSQL：从 PostgreSQL 9.5 开始支持。
- SQLite：不支持。
- DB2 LUW：至少从 1999 年开始支持。
- Oracle：从 9iR1（大约 2000 年）开始支持。
- MS SQL Server：从 2008 年开始支持。

13.1.3　使用 WITH 子句——公共表表达式

公共表表达式是一种在 SQL 语句内一次性执行东西的好方法。PostgreSQL 将执行所有的 WITH 子句并且允许在整个查询中到处使用其结果。

下面是一个简化的例子：

```
test=# WITH x AS (SELECT avg(id)
                  FROM generate_series(1, 10) AS id)
   SELECT  *, y - (SELECT avg FROM x) AS diff
   FROM    generate_series(1, 10) AS y
   WHERE   y > (SELECT avg FROM x);
  y |        diff
----+--------------------
  6 | 0.5000000000000000
  7 | 1.5000000000000000
  8 | 2.5000000000000000
  9 | 3.5000000000000000
 10 | 4.5000000000000000
(5 rows)
```

在这个例子中，WITH 子句**公共表表达式（CTE）**计算 generate_series 函数生成的时间序列的均值。其结果 x 就像一个表一样可以在查询的各处使用。在笔者的例子中 x 被使用了两次。

- 支持 WITH 子句

下面展示了哪些引擎支持 WITH 子句以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 8.4 开始支持。
- SQLite：从 3.8.3 开始支持。
- DB2 LUW：从 8（2000 年）开始支持。
- Oracle：从 9iR2 开始支持。
- MS SQL Server：从 2005 年开始支持。

注意在 PostgreSQL 中，CTE 甚至可以支持写操作（INSERT、UPDATE 以及 DELETE 子句）。

13.1.4　使用 WITH RECURSIVE 子句

WITH 子句会以两种形式出现：

- 13.1.3 节中的标准 CTE（使用 WITH 子句）。
- 在 SQL 中运行递归的方法。

CTE 的简单形式已经被 13.1.3 节所涵盖，在本节中将涵盖递归版本。

- 支持 WITH RECURSIVE 子句

下面展示了哪些引擎支持 WITH RECURSIVE 子句以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 8.4 开始支持。
- SQLite：从 3.8.3 开始支持。
- DB2 LUW：从 7（2000 年）开始支持。
- Oracle：从 11gR2 开始支持（在 Oracle 中，通常更多使用 CONNECT BY 子句而不是 WITH RECURSIVE 子句）。
- MS SQL Server：从 2005 开始支持。

13.1.5　使用 FILTER 子句

在查看 SQL 标准本身时，读者将会注意到 FILTER 子句已经在 SQL 中存在了很久（2003 年就有了）。但是，并没有很多系统真正支持这一非常有用的语法元素。

例如：

```
test=# SELECT  count(*),
               count(*) FILTER (WHERE id < 5),
```

```
               count(*) FILTER (WHERE id > 2)
       FROM    generate_series(1, 10) AS id;
 count | count | count
-------+-------+-------
    10 |     4 |     8
(1 row)
```

如果一个条件由于某个其他聚集需要该数据而不能被用在普通的 WHERE 子句中，那么 FILTER 子句就能发挥作用。

在 FILTER 子句被引入之前，同样的事情可以用一种更加麻烦的语法实现：

```
SELECT sum(CASE WHEN .. THEN 1 ELSE 0 END) AS whatever
  FROM some_table;
```

- 支持 FILTER 子句

下面展示了哪些引擎支持 FILTER 子句以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 9.4 开始支持。
- SQLite：不支持。
- DB2 LUW：不支持。
- Oracle：不支持。
- MS SQL server：不支持。

13.1.6　使用窗口函数

本书中已经深入地讨论过窗口和数据分析。因此，我们就直接跳到 SQL 兼容部分。

- 支持窗口和分析

下面展示了哪些引擎支持窗口函数以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 8.4 开始支持。
- SQLite：不支持。
- DB2 LUW：从版本 7 开始支持。
- Oracle：从版本 8i 开始支持。
- MS SQL server：从 2005 开始支持。



一些其他的数据库，例如 Hive、Impala、Spark 和 NuoDB 等也支持数据分析。

13.1.7　使用有序集——WITHIN GROUP 子句

有序集对于 PostgreSQL 是相当新的特性。有序集和普通聚集的差别在于，在有序集中，数据输送给聚集的方式确实有区别。假定用户想要在数据中找到一种趋势——数据的顺序与之相关。

下面是一个计算中位数值的简单例子：

```
test=# SELECT id % 2,
         percentile_disc(0.5) WITHIN GROUP (ORDER BY id)
   FROM generate_series(1, 123) AS id
   GROUP BY 1;
 ?column? | percentile disc
----------+-----------------
        0 |              62
        1 |              61
(2 rows)
```

只有在有排序输入时中位数才能被确定。

- 支持 WITHIN GROUP 子句

下面展示了哪些引擎支持 WITH GROUP 子句[1]以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 9.4 开始支持。
- SQLite：不支持。
- DB2 LUW：不支持。
- Oracle：从版本 9iR1 开始支持。
- MS SQL Server：必须重构查询来使用窗口函数。

13.1.8　使用 TABLESAMPLE 子句

表采样长久以来都是商业数据库提供商的实力所在。多年前，传统的数据库系统已经提供了采样。但是，这种垄断已经被打破。从 PostgreSQL 9.5 开始，我们也拥有采样问

[1] 原文是“窗口函数”，应为笔误。

题的解决方案了。

例如：

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test
   SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

首先创建一个含有 1 百万行的表。然后可以执行测试：

```
test=# SELECT count(*), avg(id)
         FROM t_test TABLESAMPLE BERNOULLI (1);
 count |          avg
-------+---------------------
  9802 | 502453.220873291165
(1 row)
test=# SELECT count(*), avg(id)
         FROM t_test TABLESAMPLE BERNOULLI (1);
 count |          avg
-------+---------------------
 10082 | 497514.321959928586
(1 row)
```

在这个例子中，从数据中取得了 1%的样本。平均值很接近于 50 万[①]，因此从统计的角度来看，该结果很不错。

- 支持 TABLESAMPLE 子句

下面展示了哪些引擎支持 TABLESAMPLE 子句以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：从 PostgreSQL 9.5 开始支持。
- SQLite：不支持。
- DB2 LUW：从版本 8.2 开始支持。
- Oracle：从版本 8 开始支持。
- MS SQL Server：从 2005 开始支持。

[①] 原文为“5 百万”，应为笔误。

13.1.9　使用 limit/offset

在 SQL 中，限制结果多少有点悲剧色彩。长话短说，每个数据库的行事方式多少有些不同。尽管在限制结果上确实有 SQL 标准，但并不是所有人都完全以那种方式支持这一特性。限制数据的正确方式实际上应该是使用下面的语法：

```
test=# SELECT * FROM t_test FETCH FIRST 3 ROWS ONLY;
 id
----
  1
  2
  3
(3 rows)
```

如果读者之前从未见过这种语法，不用担心，你并不孤独。

- 支持 FETCH FIRST 子句

下面展示了哪些引擎支持 FETCH FIRST 子句以及哪些不支持。

- MariaDB：从 5.1 开始支持（通常使用 limit/offset）。
- MySQL：从 3.19.3 开始支持（通常使用 limit/offset）。
- PostgreSQL：从 PostgreSQL 8.4 开始支持（通常使用 limit/offset）。
- SQLite：从版本 2.1.0 开始支持。
- DB2 LUW：从版本 7 开始支持。
- Oracle：从版本 12c 开始支持（使用带有 row_num 函数[①]的子选择）。
- MS SQL Server：从 2012 开始支持（使用传统的 top-N）。

如你所见，限制结果集非常难以对付，而且当用户从一种商业数据库移植到 PostgreSQL 时，用户将很可能会面对某些专用语法。

13.1.10　使用 OFFSET

OFFSET 子句是和 FETCH FIRST 子句类似的戏法。它易于使用，但是在早期并未被广泛采纳。它并不像 FETCH FIRST 子句的情况那么糟糕，但是它仍然可能导致问题。

- 支持 OFFSET 子句

下面展示了哪些引擎支持 OFFSET 子句以及哪些不支持。

[①] 原文为“finction”，应为笔误。

- MariaDB：从 5.1 开始支持。
- MySQL：从 4.0.6 开始支持。
- PostgreSQL：从 PostgreSQL 6.5 开始支持。
- SQLite：从版本 2.1.0 开始支持。
- DB2 LUW：从版本 11.1 开始支持。
- Oracle：从版本 12c 开始支持。
- MS SQL Server：从 2012 开始支持。

如你所见，限制结果集非常难以对付，而且当用户从一种商业数据库移植到 PostgreSQL 时，用户将很可能会面对某些专用语法。

13.1.11　使用临时表

某些数据库引擎提供临时表来处理版本管理。不幸的是，在 PostgreSQL 中并没有自带这种特性。因此，如果从 DB2 或者 Oracle 进行转移，用户需要做一些工作来把想要的功能移植到 PostgreSQL。说白了，在 PostgreSQL 端改一点代码并不难。但是，那确实需要一些人工介入——这已经不再是直接复制粘贴就能解决的。

- 支持临时表

下面展示了哪些引擎支持临时表以及哪些不支持。

- MariaDB：不支持。
- MySQL：不支持。
- PostgreSQL：不支持。
- SQLite：不支持。
- DB2 LUW：从版本 10.1 开始支持。
- Oracle：从版本 12cR1 开始支持。
- MS SQL server：从 2016 开始支持。

13.1.12　匹配时间序列中的模式

笔者注意到最近的 SQL 标准（SQL 2016）提供了一种特性，该特性为在时间序列中查找匹配而设计。到目前为止，只有 Oracle 在最新的产品版本中实现了这一功能。

在这一点上，还没有其他数据库供应商跟进并且增加类似的功能。如果用户想在 PostgreSQL 中模拟这种最新的技术，就必须与窗口函数和子查询打交道。在 Oracle 中匹配时间序列模式的功能相当强大，在 PostgreSQL 中没有一种查询能实现同样的功能。

13.2　从 Oracle 转移到 PostgreSQL

到目前为止，读者已经见到了如何在 PostgreSQL 中移植或使用最重要的特性。在这一介绍之后，现在可以专门介绍有关迁移 Oracle 数据库的内容。

如今，由于 Oracle 新的许可证和商业策略，从 Oracle 迁移到 PostgreSQL 已经变得非常流行。在全球，人们都在从 Oracle 离开并且采用 PostgreSQL。为了帮助那些不再使用 Oracle 的人们，笔者在这里包括了一个特别的小节。很多人已经开始转移到 PostgreSQL 以大量降低耗费在许可证上的开销。

13.2.1　使用 oracle_fdw 扩展转移数据

笔者最喜欢的一种从 Oracle 转移到 PostgreSQL 的方法是 Laurenz Albe 的 oracle_fdw 扩展（https://github.com/laurenz/oracle_fdw）。它是一种**外部数据包装器（FDW）**，它允许用户把一个 Oracle 中的表表示为 PostgreSQL 中的表。oracle_fdw 扩展是最复杂的 FDW 之一，它非常稳固可靠、有完善的文档、免费而且开源。

安装 oracle_fdw 扩展要求用户安装 Oracle 的客户端库。幸运的是，已经有一些可以直接使用的 RPM 包（http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html）。oracle_fdw 扩展需要 OCI 驱动器以便与 Oracle 进行对话。除了使用现成 Oracle 客户端驱动之外，还有一个社区提供的用于 oracle_fdw 扩展本身的 RPM 包。如果用户用的不是一个基于 RPM 的系统，用户可能必须自行编译，这显然是可行的但工作量有点大。

一旦软件被安装好，就可以很容易地启用它：

```
test=# CREATE EXTENSION oracle_fdw;
```

CREATE EXTENSION 子句把该扩展装载到用户想要的数据库中。下一步，可以创建一台服务器并且把用户映射到它们在 Oracle 端对应的用户：

```
test=# CREATE SERVER oraserver FOREIGN DATA WRAPPER oracle_fdw
          OPTIONS (dbserver '//dbserver.example.com/ORADB');
test=# CREATE USER MAPPING FOR postgres SERVER oradb
          OPTIONS (user 'orauser', password 'orapass');
```

然后就可以取得一些数据。笔者最喜欢的方法是使用 IMPORT FOREIGN SCHEMA 子句来导入数据定义。IMPORT FOREIGN SCHEMA 子句将会为远程方案中的每一个表都创建一个外部表，Oracle 端的数据就可以通过外部表披露给 PostgreSQL 的用户，这些数据之后可以被很容易地读取。

利用导入方案最简单的方法是在 PostgreSQL 上创建单独的方案，它们只保存数据库模式，然后可以使用 FDW 很容易地把数据吸取到 PostgreSQL 中。本书 13.3 节中有关从 MySQL 迁移的内容展示了如何用 MySQL/MariaDB 实现的例子。记住 IMPORT FOREIGN SCHEMA 子句是 SQL/MED 标准的一部分，因此其过程与 MySQL/MariaDB 相同。这几乎适用于每一种支持 IMPORT FOREIGN SCHEMA 子句的 FDW。

虽然 oracle_fdw 扩展为用户做了大部分的工作，但还是有必要看看数据类型是如何被映射的。Oracle 和 PostgreSQL 并未提供完全相同的数据类型，因此需要 oracle_fdw 扩展或者用户手工做一些映射。表 13-1 给出了类型如何被映射的概览，左边的列显示的是 Oracle 类型，而右边表示可能的 PostgreSQL 类型。

表 13-1

Oracle 类型	PostgreSQL 类型
CHAR	char、varchar 和 text
NCHAR	char、varchar 和 text
VARCHAR	char、varchar 和 text
VARCHAR2	char、varchar 和 text
NVARCHAR2	char、varchar 和 text
CLOB	char、varchar 和 text
LONG	char、varchar 和 text
RAW	uuid 和 bytea
BLOB	bytea
BFILE	bytea（只读）
LONG RAW	bytea
NUMBER	numeric、float4、float8、char、varchar 和 text
NUMBER(n,m)（其中，m≤0）	numeric、float4、float8、int2、int4、int8、boolean、char、varchar 和 text
FLOAT	numeric、float4、float8、char、varchar 和 text
BINARY_FLOAT	numeric、float4、float8、char、varchar 和 text
BINARY_DOUBLE	numeric、float4、float8、char、varchar 和 text
DATE	date、timestamp、timestamptz、char、varchar 和 text
TIMESTAMP	date、timestamp、timestamptz、char、varchar 和 text
TIMESTAMP WITH TIME ZONE	date、timestamp、timestamptz、char、varchar 和 text
TIMESTAMP WITH LOCAL TIME ZONE	date、timestamp、timestamptz、char、varchar 和 text
INTERVAL YEAR TO MONTH	interval、char、varchar 和 text
INTERVAL DAY TO SECOND	interval、char、varchar 和 text
MDSYS.SDO_GEOMETRY	geometry

如果用户想用几何数据，应确保数据库服务器中安装了 PostGIS。

oracle_fdw 扩展的缺点也很明显，它无法迁移内建的过程。存储过程有点特殊，它们的迁移需要一些人工介入。

13.2.2　使用 ora2pg 从 Oracle 迁移

在外部数据包装器存在之前，就已经有人在从 Oracle 迁移到 PostgreSQL。高昂的许可证开销折磨了人们很长时间，因此迁移到 PostgreSQL 也变成了一种很自然的事情。

还有一种可替代 oracle_fdw 扩展的东西名为 ora2pg，它已经存在了很多年并且可以从 https://github.com/darold/ora2pg 免费下载。ora2pg 使用 Perl 编写，并且长期都有新版本发布。

ora2pg 所提供的特性令人震惊：

- 完整的数据库模式迁移，包括表、视图、序列和索引（唯一、主键、外键和检查约束）。
- 用户和组的特权迁移。
- 分区表的迁移。
- 导出预定义函数、触发器、过程、包以及包主体。
- 完整或部分数据的迁移（使用 WHERE 子句）。
- 用 PostgreSQL 的 bytea 完整支持 Oracle 的 BLOB 对象。
- 将 Oracle 视图导出为 PostgreSQL 的表。
- 导出 Oracle 的用户定义类型。
- PL/SQL 代码到 PL/pgSQL 代码的基本自动转换。所有东西的完整自动转换是不可能实现的。不过，有很多东西可以被自动转换。
- 将 Oracle 表导出为外部数据包装器表。
- 导出物化视图。
- 显示 Oracle 数据库内容的详细报告。
- 评估一个 Oracle 数据库的迁移过程的复杂度。
- 对来自文件的 PL/SQL 代码的迁移代价评估。
- 生成可用于 Pentaho 数据集成器（Kettle）的 XML 文件的能力。
- 将 Oracle 的定位器或者空间几何导出到 PostGIS。
- 将数据库链接导出为 Oracle FDW。
- 将同义词导出为视图。
- 将目录导出为一个外部表或者用于 external_file 扩展的目录。

- 将一个 SQL 命令列表分发到多个 PostgreSQL 连接。
- 在 Oracle 和 PostgreSQL 数据库之间执行 diff 进行测试。

使用 ora2pg 乍一看很难，但实际上比看起来更加容易。基本的用法如下：

```
/usr/local/bin/ora2pg -c /some_path/new_ora2pg.conf
```

ora2pg 需要一个配置文件来运行。该配置文件包含处理该过程需要的所有信息。基本上，默认的配置文件其实已经很好了，用它可以完成大部分的迁移。在 ora2pg 的语言中，一次迁移就是一个项目。

这个配置将会驱动整个项目。在用户运行该配置时，ora2pg 将用从 Oracle 提取的所有数据创建若干个目录：

```
ora2pg --project_base /app/migration/ --init_project test_project
Creating project test project.
/app/migration/test_project/
                    schema/
                           dblinks/
                           directories/
                           functions/
                           grants/
                           mviews/
                           packages/
                           partitions/
                           procedures/
                           sequences/
                           synonyms/
                           tables/
                           tablespaces/
                           triggers/
                           types/
                           views/
                   sources/
                           functions/
                           mviews/
                           packages/
                           partitions/
                           procedures/
                           triggers/
                           types/
                   views/
```

```
                    data/
                    config/
                    reports/

Generating generic configuration file
Creating script export_schema.sh to automate all exports.
Creating script import_all.sh to automate all imports.
```

如你所见，有一些脚本会被生成，它们只需要被直接执行就好。产生的结果数据接下来就可以被很好地导入 PostgreSQL 中。不过要准备好更改各处的过程，因为不是所有的东西都能被自动地迁移，因此通常还会需要一些人工介入。

13.2.3　常见的陷阱

有一些非常基本的语法元素可以在 Oracle 中工作但可能无法在 PostgreSQL 中工作。本节列出了一些最重要的部分。当然，目前这个列表并不完整，但是它能够给读者指明正确的方向。

在 Oracle 中，用户可能会用到下面的语句：

```
DELETE mytable;
```

在 PostgreSQL 中这一语句是错误的，因为 PostgreSQL 要求用户在 DELETE 语句中使用 FROM 子句。好消息是这类语句很容易修复。

下一种可能碰到的语句是：

```
SELECT sysdate FROM dual;
```

PostgreSQL 既没有 sysdate 函数也没有 dual 函数。dual 函数部分比较容易修复，因为用户可以简单地创建一个返回单行的视图来模拟。在 Oracle 中，dual 函数的用法如下：

```
SQL> desc dual
Name Null? Type
----------------------------------------- -------- -
DUMMY VARCHAR2(1)

SQL> select * from dual;

D
-
X
```

在 PostgreSQL 中，可以通过创建下面的视图实现同样的效果：

```
CREATE VIEW dual AS SELECT 'X' AS dummy;
```

sysdate 函数也很容易对付，可以用 clock_timestamp 函数替换它。

另一种常见的问题是缺少 varchar2 之类的数据类型以及只有 Oracle 支持的特殊函数。解决这些问题的一种好的做法是安装 orafce 扩展，它提供了通常所需的大部分这类东西。因此当然有必要去看看 https://github.com/orafce/orafce 以了解更多有关 orafce 扩展的情况。它已经出现了多年并且是一种稳定的软件。最近进行的一项研究（由 NTT 完成）表明 orafce 扩展能够帮助确保所有 Oracle SQL 中的 73%能在 PostgreSQL 中不加修改地执行。

最常见的陷阱之一是 Oracle 处理外连接的方式。考虑下面的例子：

```
SELECT employee_id, manager_id
   FROM employees
   WHERE employees.manager_id(+) = employees.employee_id;
```

PostgreSQL 没有提供这类语法并且永远也不会提供。因此这个连接必须被重写为正确的外连接。加号（+）是与 Oracle 高度相关的元素，在迁移时必须把它移除。

13.3　从 MySQL 或 MariaDB 转移到 PostgreSQL

在本章中，读者已经学到了一些如何从 Oracle 等数据库中转移到 PostgreSQL 的方法。将两种数据库系统[①]迁移到 PostgreSQL 都很容易。这样做的原因是 Oracle 可能比较昂贵并且 Oracle 可能有时显得有些笨重，对于 Informix 也是如此。不过，Informix 和 Oracle 都有一个重要的共同点：CHECK 约束会被正确地对待并且数据类型会被恰当地处理。通常用户可以放心地认为这些商业系统中的数据是正确的且不会违背基本的数据完整性规则和常识。

接下来的数据库系统有点不同。我们从商业数据库中了解到的很多事情在 MySQL 中都不一样。NOT NULL 对于 MySQL 并没有什么意义（除非用户明确地使用严格模式）。在 Oracle、Informix、DB2 和所有笔者知道的其他系统中，NOT NULL 都是一条铁律，它在所有环境下都会被遵从。而 MySQL 默认并不把这类约束当回事，在迁移时这就会导致一些问题。对于技术上错误的数据我们能做些什么？如果突然发现 NOT NULL 列中有无数的空项，应该怎样处理？MySQL 并不只是在 NOT NULL 列中插入空值。它会根据数据类型插入一个空字符串或者 0。这样，事情就很令人烦恼了。

[①] 这里原文似乎有点问题，作者本意应该是指 Oracle 和 Informix 两种数据库系统，但是在此之前还未提到 Informix。

13.3.1　处理 MySQL 和 MariaDB 中的数据

正如读者可能想象或者已经注意到的，在谈论到具体数据库时笔者似乎变得不那么公正了。不过，笔者并不想把这一部分内容变成对 MySQL/MariaDB 的盲目抨击。笔者的目的实际上是向读者展示为什么 MySQL 和 MariaDB 可能在长期运行中为用户带来痛苦。笔者的偏向是有原因的，并且笔者真地想指出为什么会这样。接下来读者将看到的事情很吓人并且通常对迁移过程有严重的影响。笔者已经指出过 MySQL 有点特殊，本节将会尝试证明这一观点。

让我们从创建一个简单的表开始：

```
MariaDB [test]> CREATE TABLE data (
   id integer NOT NULL,
   data numeric(4, 2)
);
Query OK, 0 rows affected (0.02 sec)

MariaDB [test]> INSERT INTO data VALUES (1, 1234.5678);
Query OK, 1 row affected, 1 warning (0.01 sec)
```

到目前为止，没什么特殊的事情。笔者创建了一个由两列构成的表。第一列被明确地标记为 NOT NULL。第二列被假定为包含长度为四位的数字值。最后，笔者还增加了一个简单的行。不知道读者能否看到这里有一个将要被触发的地雷？很可能看不出来。没关系，检查下面的列表：

```
MariaDB [test]> SELECT * FROM data;
+----+-------+
| id | data  |
+----+-------+
| 1  | 99.99 |
+----+-------+
1 row in set (0.00 sec)
```

如果笔者没有记错，刚才增加的是一个四位数，但首先它就没有产生效果。MariaDB 直接更改了笔者的数据。当然，它发出了一个警告，但是这确实不应该发生，因为表的内容并未反映笔者实际插入的东西。

让我们尝试在 PostgreSQL 中做同样的事情：

```
test=# CREATE TABLE data (
   id integer NOT NULL,
```

```
    data numeric(4, 2)
);
CREATE TABLE
test=# INSERT INTO data VALUES (1, 1234.5678);
ERROR: numeric field overflow
DETAIL: A field with precision 4, scale 2 must round to an absolute value
less than 10^2.
```

表的创建和前面一样，但与 MariaDB/MySQL 形成鲜明对比的是，PostgreSQL 将会报错，因为我们尝试将一个不被允许的值插入表中。如果数据库引擎根本不在乎，清晰地定义我们想要什么还有何意义？假设你赢得彩票，但却损失了几百万，只是因为系统觉得那对你比较好，你会怎么想？

笔者的整个职业生涯都在与商业数据库斗争，但笔者从未在任何昂贵的商业系统（Oracle、DB2、MS SQL 等）中见过类似的事情。它们可能都有自身的问题，但是至少通常数据是正确的。

1. 更改列定义

让我们看看如果想要修改表定义会发生什么：

```
MariaDB [test]> ALTER TABLE data MODIFY data numeric(3, 2);
Query OK, 1 row affected, 1 warning (0.06 sec)
Records: 1 Duplicates: 0 Warnings: 1
```

看出这里的问题了吗？

```
MariaDB [test]> SELECT * FROM data;
+----+------+
| id | data |
+----+------+
| 1  | 9.99 |
+----+------+
1 row in set (0.00 sec)
```

如你所见，数据又被修改了。原本它就不应该在这里，而在这里又一次被更改。记住，你可能再次损失金钱或者其他一些资产，只是因为 MySQL 自作聪明。

PostgreSQL 中是这样的：

```
test=# INSERT INTO data VALUES (1, 34.5678);
INSERT 0 1
test=# SELECT * FROM data;
```

```
 id | data
----+-------
  1 | 34.57
(1 row)
```

现在让我们更改列定义：

```
test=# ALTER TABLE data ALTER COLUMN data TYPE numeric(3, 2);
ERROR: numeric field overflow
DETAIL: A field with precision 3, scale 2 must round to an absolute value
less than 10^1.
```

PostgreSQL 将再次报错并且不允许对数据做不好的事情。在任何重要的数据库中都预期会发生同样的事情。规则很简单：PostgreSQL 和其他的数据库将不允许用户毁掉他们的数据。

不过，PostgreSQL 允许用户做一件事情：

```
test=# ALTER TABLE data
   ALTER COLUMN data
         TYPE numeric(3, 2)
         USING (data / 10);
ALTER TABLE
```

用户可以明确地告诉系统如何行动。在这个例子中，笔者明确地告诉 PostgreSQL 把该列的内容除以 10。开发者们可以明确地提供适用于数据的规则。PostgreSQL 不会自作聪明，并且它有充分的理由这样做：

```
test=# SELECT * FROM data;
 id | data
----+------
  1 | 3.46
(1 row)
```

可以看到数据是符合预期的。

2. 处理空值

笔者并不想把本章变成“为什么 MariaDB 不好”，但笔者想要在这里给出最后一个例子，笔者认为它很重要：

```
MariaDB [test]> UPDATE data SET id = NULL WHERE id = 1;
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 1
```

记住，id 列被明确地标记为 NOT NULL：

```
MariaDB [test]> SELECT * FROM data;
+----+------+
| id | data |
+----+------+
| 0  | 9.99 |
+----+------+
1 row in set (0.00 sec)
```

显然，MySQL 和 MariaDB 认为空和零是相同的东西。让笔者尝试用一个简单的比喻来解释这个问题：“你知道你的钱包是空的”和“我不知道到底我有多少钱”是不一样的。在笔者写这一部分内容时，笔者不知道自己有多少钱（空 = 未知），但笔者 100%肯定不是零（笔者敢肯定足够在从机场回家的路上给笔者可爱的车加上油，但口袋里什么都没有就难办了）。

这里还有更加吓人的消息：

```
MariaDB [test]> DESCRIBE data;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   |     | NULL    |       |
| data  | decimal(3,2) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

MariaDB 确实记得列被假定为空，但它又一次更改了用户的数据。

3. 预期的问题

主要的问题是将数据转移到 PostgreSQL 时可能会有麻烦。想象一下，用户想要转移一些数据并且在 PostgreSQL 端有一个 NOT NULL 约束。我们知道 MySQL 根本不在乎：

```
MariaDB [test]> SELECT CAST('2014-02-99 10:00:00' AS datetime) AS x,
    CAST('2014-02-09 10:00:00' AS datetime) AS y;
+------+---------------------+
| x    | y                   |
+------+---------------------+
| NULL | 2014-02-09 10:00:00 |
+------+---------------------+
1 row in set, 1 warning (0.00 sec)
```

PostgreSQL 肯定将会拒绝 2 月 99 日（有充分的理由），但是它可能也不会接受空值（如果已经明确地禁止空值，也有充分的理由）。这种情况下用户必须做的是修复数据以确保它遵循数据模型的规则（由于某种原因已经存在）。用户不应该对此掉以轻心，因为用户可能不得不更改数据，而这些数据实际上原本就是错误的。

13.3.2　迁移数据和模式

在尝试解释为什么转移到 PostgreSQL 是一种好主意并且描述一些最重要的问题之后，现在是时候看看我们有些什么选项可以最终完成从 MySQL/MariaDB 迁移的工作。

1. 使用 pg_chameleon

一种从 MySQL/MariaDB 转移到 PostgreSQL 的方法是使用 Federico Campoli 开发的 pg_chameleon 工具，它可以从 GitHub 自由下载：https://github.com/the4thdoctor/pg_chameleon。该工具被明确地设计为将数据复制到 PostgreSQL 中并且为用户做很多转换模式之类的工作。

基本上，该工具会执行下面的 4 个步骤：

（1）pg_chameleon 从 MySQL 读取模式和数据，并且在 PostgreSQL 中创建一个方案。

（2）在 PostgreSQL 中保存 MySQL 的主连接信息。

（3）在 PostgreSQL 中创建主键和索引。

（4）从 MySQL/MariaDB 复制到 PostgreSQL。

pg_chameleon 提供对 DDL 的基本支持，例如 CREATE、DROP、ALTER TABLE、DROP PRIMARY KEY 等。不过，由于 MySQL/MariaDB 本身的原因，它不支持所有的 DDL，但好在它涵盖了最重要的特性。

然而，pg_chameleon 并不仅如此。笔者已经全面地阐述了数据并不总是像它应该的那样或者所预期的那样。pg_chameleon 解决这一问题的方法是抛弃垃圾数据并且把它存储在一个名为 sch_chameleon.t_discarded_rows 的表中。当然，这并不是一种完美的方案，但是对于给定的质量相当低的输入，这是笔者唯一能想到的明智的方案。其想法是让开发者决定如何处理那些损坏的行。pg_chameleon 实在没有办法决定如何处理那些被其他人损坏的东西。

近来，这个工具又进行了很多开发工作。因此，推荐用户检查该工具的 GitHub 页面并且通读所有的文档。新的特性和缺陷修复持续地被加入到该工具中，以本章有限的篇幅实在不可能完全涵盖所有的内容。



存储过程、触发器等需要特殊对待并且只能用手工处理，pg_chameleon 无法自动处理这些东西。

2. 使用外部数据包装器

如果用户想要从 MySQL/MariaDB 转移到 PostgreSQL，那么有多种方法可以完成这一工作。使用外部数据包装器是 pg_chameleon 的一种替代方案，它提供了一种快速将模式和数据导入 PostgreSQL 中的途径。连接 MySQL 和 PostgreSQL 的功能已经出现了相当长一段时间，因此探索外部数据包装器这一领域绝对对读者有帮助。

说穿了，mysql_fdw 扩展和任何其他 FDW 的效果相似。相对于其他知名度较低的 FDW 来说， mysql_fdw 扩展确实非常强大并且提供了下列特性：

- 为 MySQL/MariaDB 编写。
- 连接池。
- WHERE 子句下推（这意味着应用在表上的过滤条件可以真正地在远端被执行，这样性能更好）。
- 列下推（只有需要的列才从远端取得，较老的版本习惯于取得所有的列，这样会导致更多的网络流量）。
- 远端预备语句。

使用 mysql_fdw 扩展的方式是利用 IMPORT FOREIGN SCHEMA 语句，它允许把数据转移到 PostgreSQL 上去。

幸运的是，这在 Unix 系统上很容易做到。

要做的第一件事情是从 GitHub 下载代码：

```
git clone https://github.com/EnterpriseDB/mysql_fdw.git
```

然后运行下面的命令编译 FDW。注意在读者的系统上路径可能会不同。对于本章的例子，笔者假定 MySQL 和 PostgreSQL 都位于/usr/local 目录之下，但在读者的系统上可能不是这样：

```
$ export PATH=/usr/local/pgsql/bin/:$PATH
$ export PATH=/usr/local/mysql/bin/:$PATH
$ make USE_PGXS=1
$ make USE_PGXS=1 install
```

一旦代码被编译好，就可以把 FDW 加入数据库中：

```
CREATE EXTENSION mysql_fdw;
```

下一步是建立想要迁移的服务器：

```
CREATE SERVER migrate_me_server
   FOREIGN DATA WRAPPER mysql_fdw
   OPTIONS (host 'host.example.com', port '3306');
```

一旦服务器被创建好，就可以建立想要的用户映射：

```
CREATE USER MAPPING FOR postgres
   SERVER migrate_me_server
   OPTIONS (username 'joe', password 'public');
```

最后就是做实际的迁移。其中的第一个工作是导入模式。笔者建议首先为被链接的表创建一个特殊的方案：

```
CREATE SCHEMA migration_schema;
```

在运行 IMPORT FOREIGN SCHEMA 语句时，可以把这个方案用作目标方案，所有的数据库链接都将被存储在其中。这样做的好处是在迁移后可以很方便地删除它。

执行完 IMPORT FOREIGN SCHEMA 语句之后，就可以开始创建真正的表。最简单的方法是使用 CREATE TABLE 子句提供的 LIKE 关键词。它允许用户复制一个表的结构并且创建一个真正的本地 PostgreSQL 表。幸运的是，当正在克隆的表是一个外部数据包装器时也能这样做。例如：

```
CREATE TABLE t_customer
   (LIKE migration_schema.t_customer);
```

然后就可以处理数据：

```
INSERT INTO t customer
   SELECT * FROM migration_schema.t_customer
```

这里实际上就是用户改正数据、消除大量行或者做一点数据处理的时机。对于质量低下的数据源，在第一次转移数据之后应用约束等可能会有所帮助，这样做可能不那么痛苦。在适当的数据库中，可能有必要采用相反的过程。

一旦数据被导入，用户就可以部署所有的约束、索引等。正如笔者之前陈述的，从这里开始用户将会开始真正碰到一些让人不快的“惊喜”，所以不要指望数据是可靠的。

13.4　总　结

本章读者学习了如何转移到 PostgreSQL。迁移是一个非常重要的主题，因为就在我们谈论这些内容时，越来越多的人们已经开始采用 PostgreSQL。